Concepts and Techniques for Flexible and Effective Music Data Management

Deliege, Francois

Publication date:
2009

Document Version
Accepted author manuscript, peer reviewed version

Concepts and Techniques for Flexible and Effective Music Data Management

François Deliège

Ph.D. Dissertation

A dissertation submitted to the Faculties of Engineering, Science and Medicine at Aalborg University, Denmark, in partial fulfillment of the requirements for the Ph.D. degree in computer science.

Un jour, quand l'homme sera sage,
Lorsqu'on n'instruira plus les oiseaux par la cage,
Quand les sociétés difformes sentiront
Dans l'enfant mieux compris se redresser leur front,
Que, des libres essors ayant sondé les règles,
On connaîtra la loi de croissance des aigles,
Et que le plein midi rayonnera pour tous,
Savoir étant sublime, apprendre sera doux.
...
L'aube vient en chantant, et non pas en grondant.
Nos fils riront de nous dans cette blanche sphère;
Ils se demanderont ce que nous pouvions faire
Enseigner au moineau par le hibou hagard.
Alors, le jeune esprit et le jeune regard
Se lèveront avec une clarté sereine
Vers la science auguste, aimable et souveraine;
Alors, plus de grimoire obscur, fade, étouffant;
Le maître, doux apôtre incliné sur l'enfant,
Fera, lui versant Dieu, l'azur et l'harmonie,
Boire la petite âme à la coupe infinie.

Victor Hugo, À propos d'Horace

Abstract

The growth of digital music has yielded a high demand for applications able to organize and search in large music databases. This thesis focuses on the data management aspects that underlie modern music applications: it introduces new conceptual representations for music similarities and playlists; and it proposes effective techniques facilitating the collection, storage, search, and manipulation of music data.

The thesis begins by presenting the concept of Music Warehouses, dedicated Data Warehouses optimized for the storage and manipulation of music content. It also provides an outline of related research challenges.

A scalable framework for collecting music features and distributing the computation of music similarities without infringing copyrights is presented. The framework is evaluated on a large set of computers and is used to construct a data set of music similarities. Several methods are then introduced to improve similarity search, using weighted-combinations of collected similarities and features.

The concept of Fuzzy Song Sets is defined to flexibly capture music similarities between pairs of songs. Innovative internal representations and their corresponding implementations of the fundamental operators are studied. Bitmaps compressed with the proposed enhanced version of the Word Aligned Hybrid compression scheme prove to be well-suited for representing and manipulating Fuzzy Song Sets.

Next, the challenges bound to music search in a multidimensional space are addressed. A new bitmap compression scheme, referred to as the Position List Word Aligned Hybrid, is introduced to perform multidimensional range queries. Analytical and experimental results show significant improvements on existing bitmap compression techniques in terms of compression ratio and efficiency of bitmap operators.

Finally, the thesis defines Fuzzy Lists, a novel mathematical concept, which provides a powerful foundation for playlist manipulation. Additionally, a new compression scheme is introduced for the internal data representation of Fuzzy Lists and an implementation of the fundamental Fuzzy List operators is presented.

In conclusion, this thesis covers major issues linked to the management of music similarities and playlists and offers effective and flexible solutions that improve upon existing state-of-the-art techniques. Although the thesis focuses on the music domain, the presented techniques are general and can be applied to other domains; this substantially leverages the impact of the reported results.

List of Publications

The main body of the thesis consists of the following publications:

1. F. Deliège and T. B. Pedersen, Music Warehouses: Challenges for the Next Generation of Music Search Engines. In *Proceedings of the International Workshop on Learning the Semantics of Audio Signals, LSAS*, pages 95-105, 2006.

2. F. Deliège, Foundations of Music Warehouses for Discovering New Songs "I like". In *Proceedings of the Very Large DataBase 2007 PhD Workshop*, 6 pages, 2007.

3. F. Deliège and T. B. Pedersen, Using Fuzzy Song Sets in Music Warehouses. In *Proceedings of the 8th International Conference on Music Information Retrieval, ISMIR*, pages 21-26, 2007.

4. F. Deliège and T. B. Pedersen, Using Fuzzy Lists for Playlist Management. In *Proceedings of the 14th International MultiMedia Modeling Conference, MMM*, pages 198-209, 2008.

5. F. Deliège, B. Y. Chua and T. B. Pedersen, High-Level Audio Features: Distributed Extraction and Similarity Search. In *Proceedings of the 9th International Conference on Music Information Retrieval, ISMIR*, pages 565-570, 2008.

6. F. Deliège and T. B. Pedersen, Using Fuzzy Song Sets in Music Warehouses. In *Scalable Fuzzy Algorithms for Data Management and Analysis: Methods and Design*. Editors: A. Laurent and M.-J. Lesot. IGI Global, 28 pages, 2009.

7. F. Deliège and T. B. Pedersen, Compression of Bitmaps And Values, European patent application 09167478 US provisional patent 61/232,159

8. F. Deliège and T. B. Pedersen, Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. Submitted for publication.

Acknowledgments

A great number of people contributed to this thesis in different manners. I want to thank them all.

The work reported upon in this thesis results from close collaborations: I want to thank wholeheartedly Torben Bach Pedersen, my supervisor, and Bee Yong Chua, co-author of one of the papers.

I spent five wonderful months in Illinois as a guest of Stephen Downie in the International Music Information Retrieval Systems Evaluation Laboratory at the University of Illinois at Urbana Champaign. I would like to thank Stephen, Mert Bay, and Andreas Ehmann for their hospitality and for the numerous enlightening discussions we had. They enriched my stay, from a personal as well as a scientific point of view.

I would like to thank my colleagues in the database group in the Department of Computer Science at Aalborg University. They have always been very friendly and helpful. Without them, the long hours spent at the University would for sure have been less enjoyable.

During my stay in Aalborg, I had the chance to meet amazing people and had the privilege to become the friend of some of them. Through their outstanding contribution to my social life, they gave me the motivation and energy to pursue this work.

Finally, I would like to thank my family for their love and unconditional support. I would like dedicate this work to my parents and grand-parents.

This work was supported by the Danish Research Council for Technology and Production, through the framework project "Intelligent Sound" (STVF No. 26-04-0092).

Contents

List of Figures

Chapter 1

Introduction

The explosion of digital music technologies, such as DAB for radio broadcasting, CD for music discs, and MP3 for compressed music, has yield digital music to become the de-facto format of music distribution. Today, rare are the artists publishing their new album on a classical vinyl disc. Instead, the albums get sold through online music shops where consumers can instantly search, buy, and listen to new tracks in a few clicks. This digital revolution has drastically shifted the music distribution paradigm to become increasingly user-centric by providing personalized music recommendation to users. The challenge for music recommendation systems is to provide automated, personalized, and accurate recommendations to a large number of users on massive amount of music.

Music recommendation systems are constituted of three key elements: a *feature extractor*, a *similarity engine*, and a *Music Warehouse* (MW). The feature extractor transforms music into usable pieces of information, referred to as music features. The music features can be organized into four categories: acoustic, social, editorial, or physical features [29]. Extraction methods vary depending on the music features, e.g, a social feature extractor may collect opinions using a user interface, or may gather information from the Internet.

The similarity engine computes similarity measures between any pair of songs using the extracted features. A similarity measure can thus be created for every combination of extracted features. Typical similarity features include some parameters allowing the similarity function to be adapted dynamically, for example to fit the user's preferences.

The MW is a central repository that stores and organizes the extracted features and song similarity measures obtained from the feature extractor and the similarity engine. It facilitates the management of large music collection, for example, by enabling search for songs based on their characteristics or resemblances to other songs.

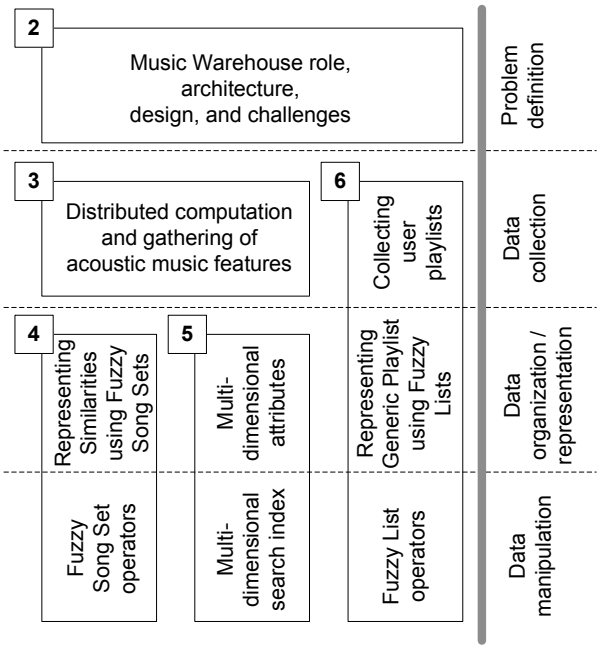


Figure 1.1: Research Topics and Related Chapters

This thesis addresses the challenges bound to the development of an MW. It relies on the existence of accurate feature extractors and music similarity engines and develops the necessary concepts and techniques required for the management of music content. The thesis reports on the development of a prototypical MW; the presented research contributions fit into its elaboration. The organization of the thesis is depicted in Figure 1.1. The topics are represented by rectangles, each topic is developed in an individual chapter corresponding to the number situated in the rectangle upper left corner.

The thesis begins by presenting the initial idea of an MW. Chapter 2 explains the motivation of constructing an MW and offers a comparison with more classical business-oriented data warehouses. It presents an overview of the different categories of musical metadata commonly used to characterize songs, the relationships between the songs and the different musical features, and proposes a system architecture. The architecture is later used as a foundation for the implementation of MW. Finally, Chapter 2 identifies ten promising challenges for the database community that are particularly beneficial in the context of music.

Features extraction is a computationally intensive process. Therefore, on a large music collection, the extraction process is typically distributed among a large num-

ber of computers. Similarly, rather than being performed in the MW, the computation of some similarity features might have to be outsourced to other computers. Chapter 3 deals with the challenges of developing a scalable framework for collecting acoustic music features. This is done by distributing the feature extraction and similarity computation tasks. The proposed framework proves to be scalable and allows thousands of computers to participate in the features extraction and similarity computations. The framework is flexible and allows similarities between songs to be precomputed or computed on the fly. Precomputed similarity measures capture the similarities between any pair of songs and do not have to fulfill any particular mathematical property, e.g., the properties of a metric. They are referred to as *relative similarity measures*. Similarity measures computed on the fly are representing each related song feature in a multidimensional space. They are referred to as *absolute similarity measures*. For both relative and absolute similarity measures, initial storage and indexing techniques are proposed. Major performance improvements of these techniques are presented in the next two chapters.

In Chapter 4, we address the issue of dealing with relative similarity measures. We present a first MW prototype able to perform efficient nearest neighbor searches in an arbitrary song similarity space. The concept of Fuzzy Song Sets (FSS), fuzzy set defined over a domain of songs, is proposed as fundamental data representation for music content in MW. Using Fuzzy Songs Sets, the MW offers a practical solution to capture user musical preferences, user feedback, and song similarities. Three approaches are considered for tackling the storage issues of FSSs: *tables*, *arrays*, and *compressed bitmaps*. After constructing theoretical estimates and developing practical implementations, results prove that, from a storage point of view, both arrays and compressed bitmaps are effective data structure solutions. With respect to speed, we show that operations on compressed bitmaps offer significant gains in performances over arrays for FSSs comprising a large number of songs. Finally, Chapter 4 explains how the presented results can be applied to other domains.

In Chapter 5, the focus is brought on improving music search based on absolute similarity measures. We propose to optimize multidimensional range queries by using bitmap indexing techniques. Compressed bitmap indexes are used for efficiently querying very large and complex databases, for example populated with music content. The Word Aligned Hybrid (WAH) bitmap compression scheme is commonly recognized as the most efficient compression scheme in terms of CPU efficiency. However, WAH compression ratios are less than optimal, especially for modern CPU architectures. Chapter 5 presents the Position List Word Aligned Hybrid (PLWAH) compression scheme that improves WAH compression significantly by making better use of the available bits and utilizing new CPU instructions. For typical bit distributions, PLWAH compressed bitmaps are often half the size of WAH bitmaps, and at the same time PLWAH has even better CPU efficiency than WAH. Using PLWAH com-

pressed bitmap indexed, storage requirements for storing the indexes are lowered and results necessitating bitmap operations are computed faster. The results are verified by theoretical estimates and extensive experiments on large amounts of both synthetic data and absolute similarity measures generated from real world music collection.

Chapter 6 introduces the notion of *generic playlists* and presents a concrete scenario to illustrate their possibilities. Generic playlists allow users to collaboratively elaborate playlist and to adapt shared playlists to fit their music preferences. Additionally, a formal foundation is provided to enable the development of playlist management tools: the concept of *Fuzzy List* (FL) is defined and a corresponding algebra is developed. A prototypical implementation of FL is presented. In order to reduce the storage requirements and improving the MW scalability, FLs are compressed using the Pattern Aware Word Aligned Hybrid (PAWAH) compression scheme. PAWAH is a new compression scheme that allows FLs operators to manipulate FL directly in their compressed format. FLs offer a flexible solution perfectly suited to meet the demands of playlist management.

Finally, Chapter 7 proposes a summary of the conclusions and future research directions. The key contributions of the thesis are summarized below. They inherently constitute pragmatic solutions as they result from the elaboration of the prototypical MW presented in the thesis.

1. Feature extraction is a CPU intensive process. On a large music collection, the extraction process is typically distributed among computers. We propose a scalable framework to distribute the extraction of the music features and compute music similarities with respect of copyrights.

2. We present FSSs and propose to use them for representing music information. An algebra for facilitating their manipulation is developed. Three usage scenarios and the corresponding multidimensional cubes extended with FSSs are described; they illustrate the usage of Fuzzy Song Sets in MW.

3. We implement common fuzzy set operators for WAH and array compressed fuzzy sets. We prove that fuzzy sets represented with WAH compressed bitmaps are an efficient approach for manipulating large fuzzy sets.

4. We introduce PLWAH, a new bitmap compression scheme, that improves the execution of multidimensional range queries commonly used for searching among music features. PLWAH offers good compression ratio and allows bitwise operations to be performed directly on the compressed bitmaps. PLWAH bitmap indexes allow efficient search on highly multidimensional data.

5. We define FLs and propose an algebra to manipulate them. FLs are a flexible concept used to represent, manipulate, and share playlists.

6. We propose a storage representation and the corresponding implementation of the operators of FLs. FLs are stored in a new compressed format referred to as PAWAH. PAWAH enables arithmetic operations on the membership degree to be performed directly on the compressed representation.

These contributions offer new interesting research directions. Future work include the integration of the feature extraction framework into new music information retrieval platforms, such as the Networked Environment for Music Analysis (NEMA) [34]. The FFSs and FLs should be integrated into new music recommendation system and dynamic playlist generator as presented in [16, 17]. Promising research on the PLWAH and PAWAH compression schemes include the development of a variable counter length and performance improvements thanks to a better usage of the multi-core architecture of modern CPUs. A brief overview of promising research directions is presented in Chapter 7.

The thesis is organized as a collection of individual papers. The papers have been polished and grouped by theme in order to be integrated in this thesis. Chapters 2 to 6 are self-contained and can be read in isolation. Since these chapters are closely related, this entails two minor overlaps: (1) some references to music databases in the related work sections of the different chapters are repeated, and (2) the brief paragraphs presenting the Fuzzy Song Sets and Fuzzy Lists in Chapter 2 are detailed in Chapters 4 and 6 respectively. The mapping between the thesis' chapters and the published papers is given below. The unpublished extensions of the papers are presented as annexes to the corresponding chapters.

Chapter 2:

- F. Deliège and T. B. Pedersen, Music Warehouses: Challenges for the Next Generation of Music Search Engines. In *Proceedings of the International Workshop on Learning the Semantics of Audio Signals, LSAS*, pages 95-105, 2006.

- F. Deliège, Foundations of Music Warehouses for Discovering New Songs "I like". In *Proceedings of the Very Large DataBase 2007 PhD Workshop*, 2007.

Chapter 3:

- F. Deliège, B. Y. Chua and T. B. Pedersen, High-Level Audio Features: Distributed Extraction and Similarity Search. In *Proceedings of the 9th International Conference on Music Information Retrieval, ISMIR*, pages 565-570, 2008.

Chapter 4:

- F. Deliège and T. B. Pedersen, Using Fuzzy Song Sets in Music Warehouses. In *Proceedings of the 8th International Conference on Music Information Retrieval, ISMIR*, pages 21-26, 2007.
- F. Deliège and T. B. Pedersen, Using Fuzzy Song Sets in Music Warehouses. In *Scalable Fuzzy Algorithms for Data Management and Analysis: Methods and Design*. Editors: A. Laurent and M.-J. Lesot. IGI Global, 30 pages, 2009. To appear.

Chapter 5:

- F. Deliège and T. B. Pedersen, Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. Submitted for publication.

Chapter: 6

- F. Deliège and T. B. Pedersen, Using Fuzzy Lists for Playlist Management. In *Proceedings of the 14th International MultiMedia Modeling Conference, MMM*, pages 198-209, 2008.

Chapter 2

Music Warehouses: Challenges for the Next Generation of Music Search Engines

Music recommendation systems have recently become very popular. While their first generation was only based on a few manually extracted musical descriptors, the new generation will provide richer and more accurate metadata over very large volumes of music and will be able to generate personalized playlists automatically. These new major improvements call for the development of innovative data management techniques. This chapter proposes to satisfy this need by extending technology from business-oriented *Data Warehouses* to so-called *Music Warehouses* (MW) that integrate a large variety of music-related information, including both low-level features and high-level musical information. Our work on MWs focuses on nearest neighbor searches and the generation of personalized user playlists. Finally, this chapter presents a number of other new challenges for the database community that must be taken up to meet the particular demands of music warehouses.

2.1 Introduction

The tremendous growth of digital music available on the Internet has created a high demand for applications able to organize and search in large music databases. Thanks to new digital music formats, the size of personal music collections often reaches up

to thousands of songs and large online music stores and online radios are becoming very popular. However, current search tools still remain very limited: popular search engines only provide searches based on external annotations, but to offer truly natural and intuitive information retrieval, search and query into the primary media is required. These needs have given further impulse to the development of new methods for music information retrieval and research on digital music databases.

Companies have always spent a considerable amount of money and efforts to ensure proper storage and management of their business information in order to answer questions about sales, production, or any operation relevant to their particular business concerns. Therefore, in large companies, each operational unit has always gathered, on a regular basis, various pieces of knowledge using a number of systems. Unfortunately, these systems have usually been provided by different vendors over a long period of time and are based on different technologies and terminologies which often make integration a major problem. This integration is, however, needed when it comes to answering questions implying data from different operational units. For example, in order to determine the profitability of a given product, data from sales and production needs to be combined. Another example is trend analysis that requires combining the budget and the performance information over time. To solve this centralization problem, the *data warehousing* approach integrates data coming from the various operational units into one common data store, referred to as the *data warehouse* (DW), optimized for data analysis purposes.

The data warehousing approach has already demonstrated its strengths in the business context and has been widely used as a solid ground for *On-Line Analytic Processing* (OLAP) systems. OLAP systems allow queries such as calculating the profitability of products categories over the years to be answered "live". At the same time, such systems, regardless of the database management system used, have commonly adopted the same conceptual *multidimensional view of data*.

In other contexts, however, applications call for more *complex data structures* than the ones proposed in the classical *multidimensional data model*. One such domain is *music classification and retrieval*. Automated classification of song descriptors, computer or manually generated as in the Music Genome Project [1], has already received a lot of attention from the signal processing and machine learning research communities as well as from private companies. Also, the database community has shown an increasing interest in creating new indexes able to search among large amount of complex data such as music content descriptors. However, to the best of the authors' knowledge, no work has been reported so far concerning the management of musical information using multidimensional models.

Music Warehouses (MWs) are dedicated DWs optimized for the storage and analysis of music content. They provide the advanced framework necessary to support

[1] http://www.pandora.com/mgp.shtml

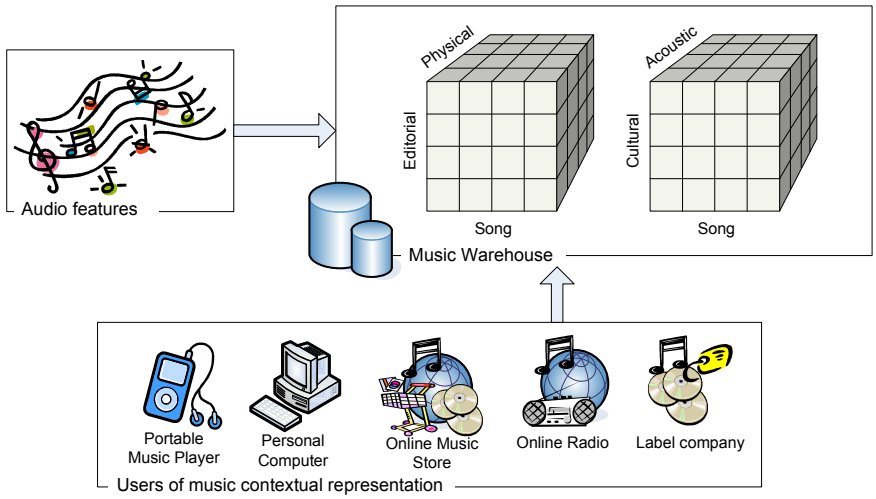


Figure 2.1: The Key Role of an MW

semantic tools able to close the gap between audio features and contextual representations. In particular, the multidimensional view of data commonly adopted in DWs facilitates the understanding of the mapping between low-level features and high-level representations. Also, the summarization features integrated in multidimensional models present a prominent advantage. As pictured in Figure 2.1, MWs will play the key role of centralizing and integrating all music information pieces together. In order to capture the context of a song, MWs will use an advanced data model and its query language. Thanks to specifically designed query optimizations fast responses time will be ensured. The unequaled amount of music information available through MWs will be accessible to a large variety of clients, from personal music players to large music label companies.

The main focus of this chapter is to identify and describe the various new challenges to multidimensional database models in the music classification field. The music world requires more powerful data model constructs than the ones offered by traditional multidimensional modeling approaches. However, the issues discussed here are not confined to the music domain but will find applications in other contexts.

2.2 DW Background

The term “data warehouse”, first used by Barry Devlin [31], is best defined by Bill Inmon who explains the concept as follows: “A data warehouse is a subject oriented,

integrated, non-volatile and time-variant collection of data in support of management's decisions" [46]. A brief explanation of the different properties follows.

- *subject oriented:*
  Data is traditionally organized to support specific processes, i.e., the same data might be organized very differently depending on the process for which it has been stored. For example, it is likely that user profiles in a online music store is organized differently than on a play-list generator application of a online radio station. In DWs, data is organized by subjects rather than by processes, i.e., the same data organization is used regardless of the analysis performed.

- *integrated:*
  A classical approach for handling complex problems is to split them into smaller ones, each of them using its own structures, defining its own processes and using its own data storage format. To the contrary, in DWs, data is gathered from a variety of sources and merged into a coherent whole. In the music context, this implies that all aspects of music must be uniquely defined, a challenging task given the many inconsistencies often found in music ontologies.

- *non-volatile:*
  In process centric applications, data is often kept only for a relatively short period of time, e.g., 3 to 6 months, as these applications are mostly interested in short time horizon events. In data warehouses, however, data is kept for longer period of time in order to perform data analysis. For example, queries such as "what is the evolution of the most popular rhythm in rock music from its beginning to the 90's using 5 years time windows?" could be performed as most DWs are keeping data for years.

- *time-variant:*
  Time is one of the basic construct of a data warehouse. Frequently represented by its own dimension, the tight integration of time in DWs allows data to evolve over the time, i.e., data can be updated and changes can be tracked. In DWs, data is marked to be valid given certain time windows. So, even if Sting played in various bands during his career, it is possible to know when he was playing in Police.

- *management's decisions:*
  The fact that data warehouses are designed and optimized for query and information retrieval and not for data entry is well captured by the word "decision". First designed as a high level decision support tool at a strategic level, data warehouses have proved their usefulness at many decision levels. Today, the expression "management's decisions" is outdated. Instead, the term

> "knowledge worker" has replaced the word management, as decision points are pushed down in the organizational hierarchy. This is particularly true in the music context where music labels, music retailers, copyright companies, radio stations and consumers will all interact with MWs.

DWs first appeared in the business context to integrate data from different operational units together and provide complete understanding and better coverage of the business matters. Driven by the market, academic research quickly showed interest in the topic. A major focus from both worlds has always been to support OLAP functionalities together with good performance. Research was performed on both conceptual and physical levels and has led to the creation of many different multidimensional models and OLAP systems. Multidimensional models can be divided into 3 categories: *simple cube* models, *structured cube* models and *statistical object* models [76]. OLAP systems have mainly been implemented using two technologies: *Relational OLAP* (ROLAP), based on a *Relational DataBase Management Systems* (RDBMS) [53], and *Multidimensional OLAP* (MOLAP), based on a dedicated *Multidimensional DataBase Management Systems* (MDBMS) [87]. Proper identification of the requirements of music classification systems is a first step to determine which conceptual and physical data warehouse elements are the best suited to take up the challenges offered by MWs.

2.3 Related Work

Today, most of the music recommendation systems available online are solely based on editorial information available through tags or on collaborative filtering. With the growth of the Web, techniques based on publicly available data have emerged. They combine data from many sources using text analysis and collaborative filtering techniques to determine similarity. Since they are based on human opinion, these approaches capture many cultural and other intangible factors. A main disadvantage of these techniques, however, is that they are only applicable to music for which a reasonable amount of reliable public information is available. MoodLogic[2] is an example of how editorial information and cultural information can be gathered. The core idea of MoodLogic is to associate metadata to songs automatically, thanks to two basic techniques: first, an audio fingerprinting technology able to recognize music titles on personal hard disks, second, a database collecting user ratings on songs, which is incremented automatically in a collaborative fashion. No acoustic analysis of the song is, however, performed. Pandora[3] adopts a radically different approach

[2] http://www.moodlogic.com
[3] http://www.pandora.com

and bases its recommendations on acoustic metadata. The acoustic metadata is generated through the music genome project that consists of a small group of music experts manually describing the audio content of songs. While these systems provide useful services for Electronic Music Distribution (EMD) systems, their scalability remains limited as they rely only on expensive and external sources of information.

Many researchers have studied the music similarity problem by analyzing symbolic representations such as MIDI music data, musical scores, and the like. For example, methods have been developed to search for pieces of music with a particular melody. The queries can be formulated by humming and are usually transformed into a symbolic representation, which is matched against a database of scores, e.g., in MIDI format [42]. However, techniques based on MIDI or scores are limited as they are format dependent. Acoustic representation allows music content to be directly analyzed and can, therefore, be applied to any music. A considerable amount of work has been reported on automatic extraction of audio features [44]. For monophonic music, Downie indexed with good results a database of 10000 songs by adapting existing text information retrieval techniques to music [36]. Blum et al. [14] present an indexing system based on matching features such as pitch, loudness or Mel-frequency cepstral coefficients, briefly MFCC. Tzanetakis extracts a variety of features representing the spectrum, rhythm and chord changes and concatenates them into a single vector to determine similarity [89]. Aucouturier and Pachet model songs using local clustering of MFCC features, determining similarity by comparing the models [10]. Using the extracted features, attention was drawn to enable music lovers to explore individual music collections [59,60]. Within this context, several research projects have been conducted in order to pursue a suitable similarity measure for music [63, 68].

Research on distributed music database in P2P and Wireless Ad-hoc networks was conducted by Karydis et al. [51]. They also have developed an algorithm that efficiently retrieves audio data similar to an audio query. The proposed method utilizes a feature extraction technique for acoustical music sequences [52]. Downie et al. [35] describe a secure and collaborative framework for evaluating music information retrieval algorithms but little attention has been paid so far to the storage issues of audio features in data warehouses. A more traditional approach is to use classical relational models such as the one proposed by Rubenstein that extends the entity-relationship data model to implement the notion of hierarchical ordering, commonly found in musical data [79]. Through some examples, Rubenstein illustrates how to represent musical notation in a database using the extensions he introduces, but no detailed data types and operations are given. A multimedia data model, following the layered model paradigm that consists of a data definition layer, a data manipulation layer, a data presentation layer, and a control layer, is presented by Wynblatt et al. [95], but no query language is proposed. Surprisingly, none of these models adopts a multidimensional approach by representing data in cubes, a very convenient

structure for performing on-the-fly analysis of large volumes of data that has already proved its strengths in data warehouses [72, 73]. Finally, the most relevant related work is the music data model and its algebra and query language presented by Wang et al. [90]. The data model is able to structure both the musical content and the metadata but does not address performance optimization issues. In particular, it does not provide an adequate framework to perform similarity based search, dimensions do not support hierarchies, and no discussion is provided about the indexing issues.

Existing indexing techniques, such as M-grid [32] or M-tree [25], can be applied to index high dimensional musical feature representations. However, as a consequence of the subjective nature of musical perception, the triangular inequality property of the metric space is typically not preserved for similarity measures [13,58,74]. Therefore, additional techniques have to be employed to ensure a suitable foundation for musical similarity search. Nearest neighbor searches are a popular topic in the database community for their usage in content based retrieval and similarity searches. An impressive amount of work can be found in the literature for both high and low dimensional spaces [54, 81, 91, 97]. However, these techniques are inspired by geometric problems and rely on the existence of a Euclidian space in order to eliminate potential candidates using upper and lower bounds. Unfortunately, they are not suitable for non-metric space and are therefore not applicable for the similarity measures where the triangular inequality property is not fulfilled. Work on indexes for non-metric space is presented in the literature [43, 80, 82, 96]. Though the similarity function is non-metric, it remains confined in a pair of lower and upper bounds specifically constructed. However, we aim to create MWs able to work with any similarity function as no consensus has yet been reached among the MIR community.

2.4 Musical Classification

2.4.1 Musical Metadata

The music industry needs musical classification. While various classifications exist, no real consensus seems to have emerged. Music retailers, music labels, copyright companies, radio stations, end users, etc., have all designed their own taxonomies. Music retailers taxonomies, for example, that are aimed at guiding consumers in shops, are made up of four levels alphabetically ordered: global musical categories, subcategories, artist names, and album names. Even among the same group of interest, e.g., online music portals, inconsistencies are easy to find. One notable source of inconsistencies is the use of different kinds of *metadata*, e.g., Moodlogic, Amazon and Pandora that are using different recommendation systems based respectively on editorial, cultural, and acoustic metadata, end up with end up with different classifications.

Metadata is commonly used in the research field of audio mining covering areas such as audio classification and retrieval. Literally "data about data", metadata is defined as information about another set of data. In his work on musical knowledge management [66], Pachet classifies musical metadata into three categories depending on the nature of the source from where the information can be extracted. We propose a division in four categories presented below.

- In the audio context, metadata elements such as the title, the composer, the performer, the creation date and the publisher of a song are the most commonly used. They are referred to as *editorial metadata* and give authoritative information provided mostly manually by experts. Editorial metadata covers a wide range of information, from administrative to historical facts.

- *Cultural metadata* is defined as knowledge produced by the environment or culture resulting from an analysis of emerging patterns, categories or associations from external sources of documents. Typical methods for generating such information are to use radio station play-lists to find correlations between songs or to crawl music web sites to gather word associations. An example of cultural metadata is the list of the most common terms associated with a given artist. Many online music stores, e.g., Amazon.com, are using cultural metadata based on user recommendations, a well-known collaborative filtering technique.

- *Acoustic metadata* is the third category of music information. Acoustic metadata is defined as purely objective information obtained solely through an analysis of the audio content of the music. However, acoustic metadata remains dependent on the internal primary support of the musical information. While numerous approaches exist on what acoustic features to retain and how to select these features, they can primarily be separated into two classes: symbolic representation (MIDI) and acoustic representation (WAV, MP3). In the symbolic representation, the features usually refer to pitch, rhythm, or their variations, while in the acoustic representation the most common features are produced by time analysis, spectral analysis and wavelet analysis.

- *Physical metadata*, a new fourth category of metadata, is defined as information directly related to the medium holding the music. Contrarily to cultural metadata, physical metadata is not produced by external elements such as the culture but rather provides information on the physical storage characteristics and its related practical constraints. A naive example would be the location of a music file on a computer, a possibly helpful piece of knowledge about the user's classification. Physical metadata includes information such as the type of medium, e.g., a CD or a vinyl record, the kind of media, e.g., a music or a

video clip, the format of the source, e.g., PCM or MP3, the compression used, e.g., lossless or lossy compression, etc. Physical metadata contains a lot of useful information in the context of an online music store where, for example, customers, depending on the speed of their internet connection, might want to buy video clips, music with high sound quality or music with lower sound quality.

Together, the four categories of metadata reflect the song context that can only be captured by a large quantity of metadata, possibly of high dimensionality and using heterogeneous units. Along with musical descriptions, methods to uniquely identify pieces of music are needed. Various robust audio fingerprint techniques have been developed to allow audio identification of distorted sources. Such techniques have already been successfully implemented in some systems such as the Moodlogic Music Browser[4].

2.4.2 A Case Study of Musical Management

The case study illustrates the special demands of MWs. An ER diagram of the case is shown in Figure 2.2 using the notation of [38]. It pictures at a conceptual level the data model and is, therefore, not represented using a star-schema.

The *song* is the most important entity type, as indicated by the placement in the center of the diagram. A song is uniquely defined with an song identifier (SID) and has additional attributes such as Title, Length, Format, all of which are considered to be static. Audio fingerprints allow the song to be identified uniquely based on its audio content and independently of its storage format. A song has many relationships with other entities, whose purposes are to describe the song. These other entities might be viewed as *dimensions* and are shared by all songs.

First, a song can be characterized by its editorial information. Each song is authored by one or more composers and can be played by one or more performers. Both composers and performers are artists and are identified using their scene name along with some biographic elements. Performers usually form bands together. Each band is identified with a name and has at least one time interval in which it existed. Performers may join or leave the band without the band being dissolved. Bands are able to dissolve and reunite multiple times. A song is published by a music editor at a given time, either in a single or in album identified by a name, using distribution channels such as web radios, music television channels, online music stores, etc.

Second, using collaborative filtering and user profiles, the cultural context surrounding a song can be depicted. Co-occurrence analysis is performed by tracking user play-lists and by crawling the web [27]. Each time a song is played, the previously played song is stored, so that the list of a user's most frequently played songs

[4] http://www.moodlogic.com/

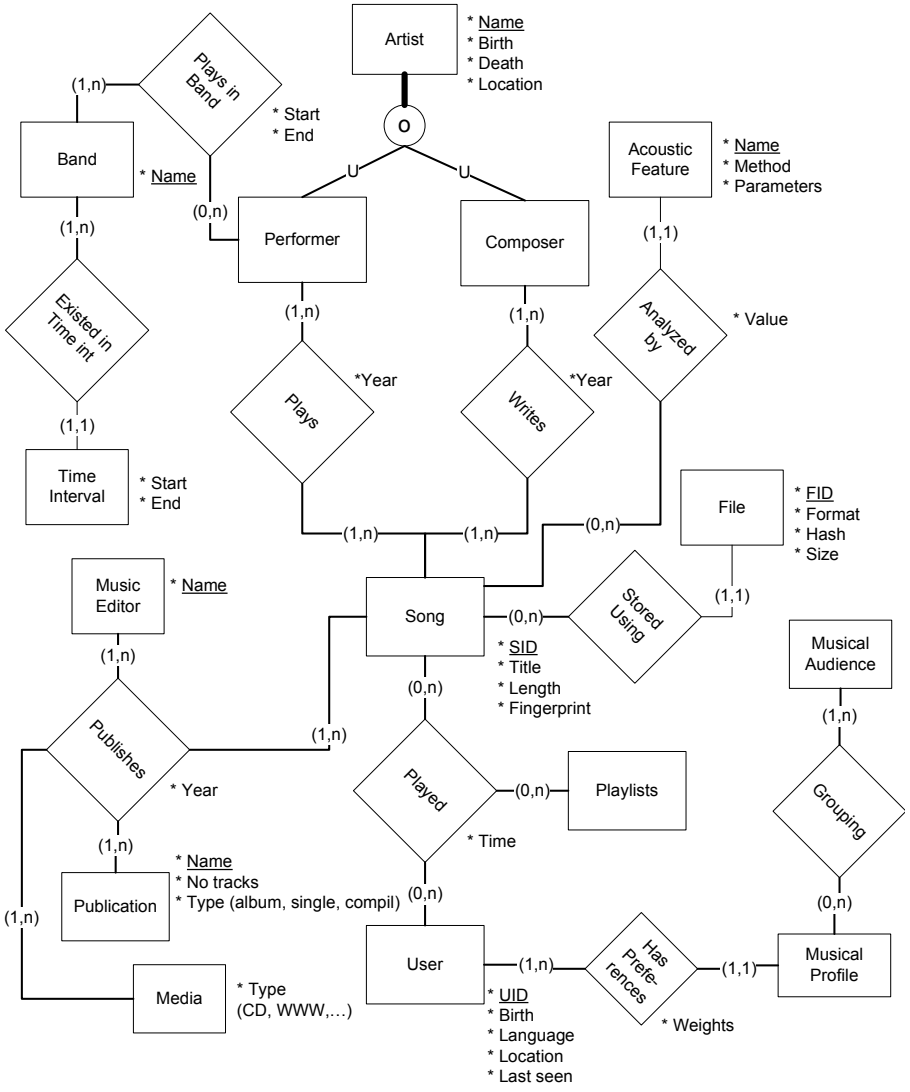


Figure 2.2: Case Study of an MW

after a given one can be inferred. A user is uniquely identified using a user identifier (UID). Each user has a location, a date of birth, a language attribute, and possibly a music profile. The music profile stores which descriptors a user values the most. A music profile is defined on a user basis and is composed of a list of weights corresponding to the music descriptors. Based on the user music profiles, groups of similar profiles can be formed. Music profile groups link users that seem to have the "same ear", i.e., using the same criteria. Similar musical profiles, i.e., users identified as having the same musical tastes, can be grouped together into *musical audiences*.

Third, a song is described with acoustic information. For each song, acoustic features can be extracted using an extraction method and its parameters. Each acoustic feature is characterized by a unique pair of an extraction method and its parameters.

Finally, each song is stored using at least one physical medium, e.g., a file where the sound data has previously been encoded in one of the well known encoding formats such as MP3, WMA, or OGG. Each file is given a unique file identifier (FID), and is characterized by an identification tag such as a hash-key that permits to search if a file is already present, an audio format representing the encoding format of the sound, its size, its quality, etc.

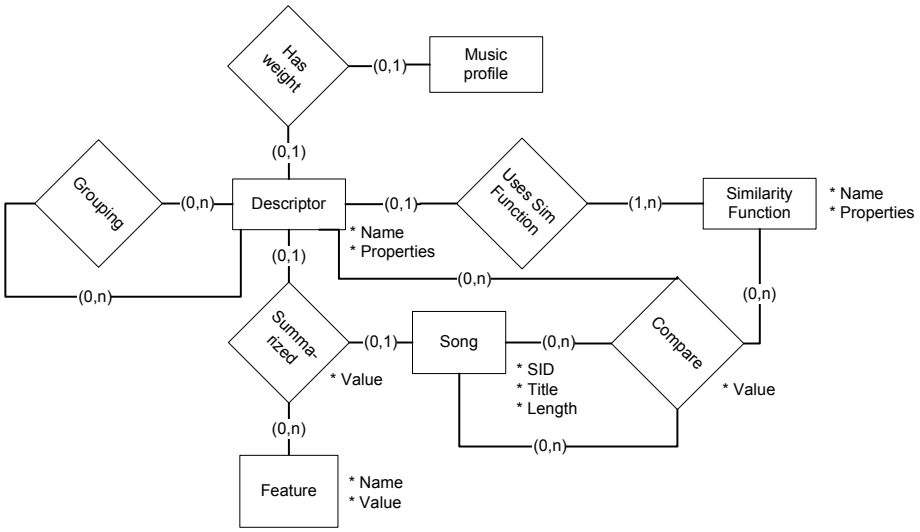


Figure 2.3: High Level Features in an MW

Figure 2.3 presents the music features at a higher abstraction level. The various features capturing the song context are all considered as descriptors regardless of the category of musical metadata they belong. Descriptors are represented in the

center of the figure as they are the most important entity type. Each descriptor is identified uniquely by its name. Multiple descriptors can be grouped together to form *fused* descriptors. Each song should be represented using as many descriptors as possible. Each descriptor has a weight reflecting its importance to each user. Finally, each descriptor should have at least one similarity function attached to it. Similarity functions allow comparison between values of a given descriptor for different songs. Once the descriptor similarities between two songs have been calculated, they can be computed into a general similarity value using the user weights.

2.5 Our approach

2.5.1 Scenario

We propose to develop a on-line music recommendation system able to find the song the most similar to another with respect to one or a combination of musical aspects. The musical aspects should represent high level metadata accessible to users such as the beat or the genre of a song. They should not only be based on audio analysis but also on cultural elements such as the targeted audience, on editorial aspects such as the artists, or on physical information such as the type of format. Furthermore, the MW has to handle user playlists and offer playlist manipulation tools. Eventually, we want to offer to users an automatic music playlist generation service. The playlist should be created with respect to constrains such as the user's profile. Additionally, the user could specify the length of the playlist and how the playlist should evolve.

2.5.2 Architecture Overview

Figure 2.4 shows our prototypical system architecture. The system is composed of a client and a server. The client is responsible for extracting new music features, for building the user profile, and for all the user interface related issues. The server is responsible for centralizing music information, for storing the user profile, for finding similar songs, and for generating playlists. First, the music not yet present on the server gets analyzed and the low level features are extracted and transmitted to the server. Based on the low level features, computationally expensive similarity value are generated. Second, once the data cubes are generated, queries to obtain the songs the most similar to a given seed song or to generate a playlist can be answered. User feedbacks on the provided results are saved for a future integration in the MW.

2.5.3 Song Similarities

The music recommendation is able to compare and provide a similarity value for any pair of songs. Using the similarity values, it is easy to tell if the two songs of any

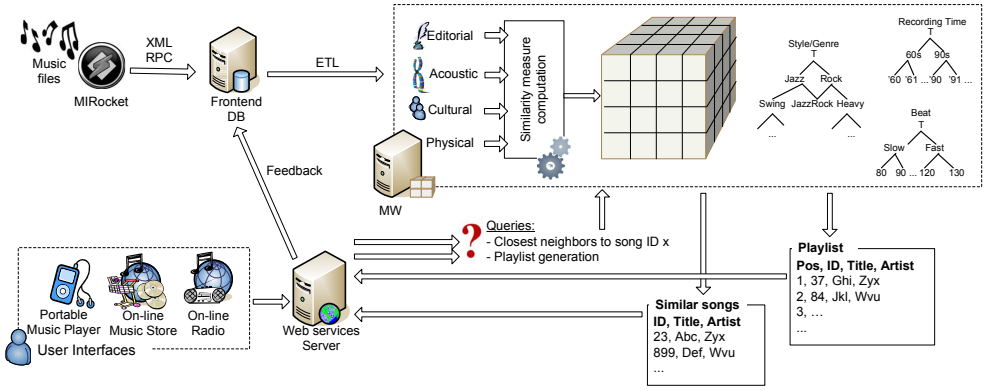


Figure 2.4: Architecture Overview

given pair of songs are "very different", "different", "somewhat similar", or "very similar" from the perspective of a given attribute such as the beat for example. But, in order to keep our scenario as general as possible, as few assumptions as possible should be made about the properties of the similarity functions. In particular, the similarity aspects do not have to be a metric in a mathematical sense (positive, triangular inequality, and symmetry properties). However, we assume that the similarity values can be mapped to values in the interval from 0 to 1.

For each song, referred to as the seed song, the similarity values with the other songs are computed and stored in a fuzzy set. The similarity value between two songs is then represented using a membership degree, e.g., the more similar two songs are the closer to 1 the membership degree will be. To retrieve the closest songs to a given song seed, the MW reads the fuzzy set and returns the songs having the highest membership degree. While this solution offers fast retrieval of the most similar songs regardless of the properties of the similarity functions, it also suffers from major drawbacks namely the storage consumption and the updating process.

We are currently extending the fuzzy set algebra presented by Galindo et al [41]. We have created operators to perform Top_k selections, reductions, and average aggregations among fuzzy sets. The Top_k operator changes the membership degree to zero to all elements that are not part of the k elements having the highest degree. The reduction operator changes to 0 all membership degrees below a given threshold. As an example, Figure 2.5 shows the aggregation of two song sets using the intersection operator. Using the fuzzy set algebra, it is possible to execute complex queries such as retrieving the songs that are the most similar to two given song seeds, or to one song seed with respect to different similarity aspects.

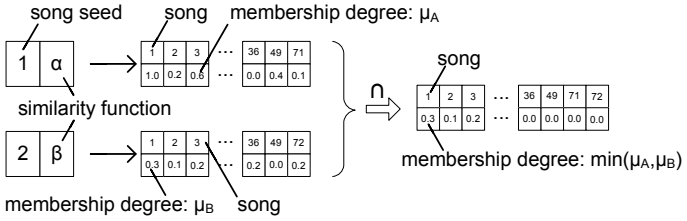


Figure 2.5: Intersection of Fuzzy Sets of Closest Songs

Additionally, we are now focusing on query optimization aspects of these operators for the different physical storage options. We have studied the use of tables, arrays and WAH compressed bitmaps [92] as internal storage solutions for the fuzzy sets and discussed the advantages of each. Tables are space consuming but allow pivoting between songs seeds and songs in the fuzzy set. The choice between arrays and compressed bitmaps depends on the number of bits required to identify uniquely an element of the fuzzy set and to store the membership degrees. The choice is, therefore, depending on the data set.

We envision the use of fuzzy sets in two additional cases. First, fuzzy sets can be used to represent users' opinion about the previously suggested songs. It is possible to retrieve the set of the proposed songs which a user liked or disliked in a particular session. Second, regardless of the context, e.g., the previously played songs, the suggested songs, or the criteria used, a user is able to grade if he likes a song or not. For example, a song banned by the user should never be played. To the contrary, other songs should be proposed more often as the user likes them. The list of songs a given user likes, and the songs he dislikes can also be stored using a fuzzy set. The MW allows aggregation to be performed using the favorite songs fuzzy set attribute of the user dimension, e.g., counting the users having marked a particular set of songs as their favorite music.

The closest songs cube provides a set of songs which are the closest to a given one, referred to as the seed song, with respect to a similarity function, so that, for each song, for each similarity function, the closest songs are stored using a fuzzy set. It is composed of the song and the similarity dimensions as shown in Figure 2.6. The user feedback cube gathers relevance statistics about the songs proposed to users by the music recommendation system. It is composed of the user dimension and the query dimension. For each user and query, the user feedback is stored. The feedback given for a particular song played is stored as a membership degree representing how the proposed song is pertinent in the context of the query. A very low membership degree is given when the user believes the song should not have been proposed.

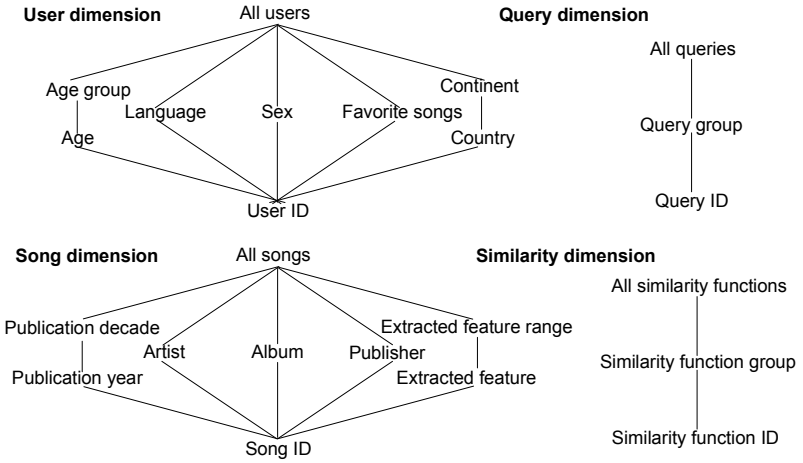


Figure 2.6: Dimension Hierarchies

2.5.4 Generic Playlists

We want to build, through user collaborative filtering, playlists composed of a given number of songs and a theme. For example, 100 users have to build a playlist that is made of ten songs, that is composed of 3 rocky songs, 3 jazzy songs, 3 romantic songs and 1 blues song. Furthermore, the users are asked to respect some smooth transitions between the songs so that the third rocky song sounds a bit jazzy. Finally, let's also assume that independently from the playlist building process each registered user has a list of songs he likes and dislikes.

The playlists created by the voting users are merged into a unique playlist, referred to as a *generic playlist*, that can be shared among all the users registered in the system. However, some users might be disappointed by the generic playlist if inserted blindly to their music player. For example, the user might have previously banned some songs and would prefer the system to find alternatives. To the contrary, if a song was very close from being selected as part of the playlist and the user rated it as his favorite song, the system should switch the song originally present in the generic playlist with the user's favorite song. Figure 2.7 illustrates the construction of a generic playlist and how it can at be derived into a personalized playlist a later stage.

Generic playlists offer a concrete scenario for the usage of *fuzzy lists*, the fuzzy counterpart of the well-known (crisp) lists. We propose to define a finite fuzzy list, A, of size m, over a domain of discourse, denoted X, as follows:

$$A = \{\mu_A(x,n)/n/x : \\ x \in X, n \in \{1,\ldots,m\}, \mu_A : X \times \mathbb{N} \mapsto [0,1]\}$$

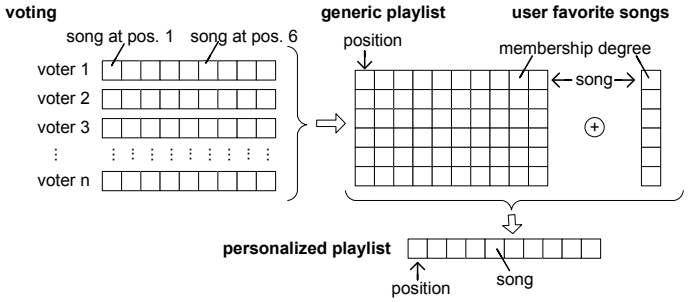


Figure 2.7: Generic Playlists Usage Scenario

where x is an element of X, where n is a non-negative integer, and where $\mu_A(x, n)$, referred to as the sequential membership degree of x at position n, is a real number belonging to $[0, 1]$, with $\mu_A(x, n) = 0$ when x does not belong to A at position n, and $\mu_A(x, n) = 1$ when x completely belongs to A at position n. The fuzzy list concept is new and new operators such as the concatenation, the intersection, the union, the subset, the inverse, and the negation have to be formally defined and studied. We believe fuzzy lists are a superset of both fuzzy sets and classical crisp lists theories, they could therefore be used in a large scope of applications. We are currently conducting further investigations on the storage and the query processing issues fuzzy lists introduce.

2.6 Challenges for MW

One of the most prominent demands for MWs is the creation of a data model supporting more complex modeling constructs than classical multidimensional models, while keeping their strengths for decision support, i.e., including the full generality of ER models would be a turn back. The data model should provide integrated semantic support for the demands that follow.

1. *Time series:*
   Many acoustic descriptors, such as the beat or the pitch, can be represented as multidimensional vectors at successive time points. Unlike typical DW facts, these types of data clearly yield no meaning when summed up. Other standard aggregation operators such as MIN, MAX and AVG do apply, but real demands are for more complex operations, such as standard deviation and other statistical functions that are useful for the similarity functions that underlay music queries. The data model should include operators allowing to cut, add, and compare time series along with aggregation operators enabling modifications

of the sampling frequency of the time series. Furthermore, the model should support irregular time series in which samples are separated by non-uniform time intervals. Finally, it should be possible to use the above-mentioned advanced temporal concepts wherever meaningful.

2. *Standards compatibility:*
   Many different formats are used to store music. While acoustic formats, e.g., MP3, OGG, WAV, contain information about the audio wave transmitted, symbolic formats, e.g., MusicXML, Humdrum, Guido, represent high level encoding information such as the duration and the intensity of the notes. The current trend for representing audio content description is to use the symbolic MPEG-7 standard in XML format [48]. The MW should be able to integrate a number of different standards such as MPEG-7 and capture data into its multidimensional model.

3. *Data imperfections:*
   In addition to the editorial, acoustic and cultural metadata, physical metadata, such as sampling frequency and format, could also be integrated in an MW to provide knowledge about the source quality. For example, a statistical measure of correctness could be applied to the title of songs with regards to where the information comes from, e.g., an original CD, a peer-to-peer sharing network, or simply missing information. Furthermore, given the large variety of music formats that support audio content, all automated extraction methods may not always be applicable or may apply with various degrees of precision, creating imperfections into the MW descriptors. Together, physical information and knowledge of imperfections should enable quality-of-service in MWs.

4. *Precision-aware retrieval:*
   Certain queries performed in an MW do not require exact answers. Rather, rough approximations would be sufficient. For example, nearest neighbor queries, such as the ranking of the k nearest neighbors of a given song, do not focus on the exact position of each song compared to a given one, but rather on coarser notion of distance, such as very close, close, or far. The exact granularity of the answer should not be fixed but rather determined either implicitly by an appropriate algebra, or explicitly in the query. Queries including the notion of ranking, referred to as Top-K queries, are very frequent in data warehouses. At the query processing level, optimizations can be performed in order to drastically improve the response time. Operators such as ranked selection and ranked joins use specific algorithms that have already demonstrated their usefulness for relational models.

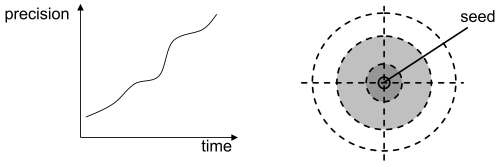


Figure 2.8: Precision Aware Retrieval

In MWs, however, Top-K queries require ranking at a coarse level of granularity where elements need to be ordered only in subsets, e.g., very close, close and far. Another aspect of quality-of-service in MWs is the response time. Time consuming queries, such as music comparison and nearest neighbor searches, spark the need for new techniques able to trade fast response time for precision. Query answers need to take the form of streams, updated progressively with more precise and reliable information as pictured in Figure 2.8.

Our contribution will consist in developing an algebra and query processing techniques to enable both coarse Top-K queries and *precision improvement streams*. For example, asking what the common characteristics of a set of songs are could result in the immediate creation of a stream of music characteristics in their high-level representation, starting with coarse similarities and progressively refining similarities as the query processing continues.

5. *Many-to-many relationships:*
In traditional multidimensional models, facts are linked to the base elements of the dimensions using one-to-many relationships. Three classic alternatives exist to encode many-to-many relationships using multidimensional modeling: traditional dimensions, mini-dimensions, and snowflaking. Using traditional dimensions, all the possible combinations of artists are created. Since the number of combinations grows at an exponential rate when adding artists, this solution quickly becomes infeasible. Limiting the enumeration to only the combinations actually used still leads to a large number of dimension records. Using mini-dimensions with one dimension for each possible artist will lead to a large number of dimensions, causing performance problems. Finally, snowflaking offers no advantage over traditional dimension as the number of basic elements would remain equal. Classical multidimensional models are able to capture the fact that an artist can perform many different songs but not the fact that multiple artists can perform together in a single song. Counting how many titles where performed by either artist A or B, becomes a dreadful task if we consider that songs performed by both artists should only be counted once. Instead, the intended behavior should be directly captured by the schema.

6. *Versioned irregular hierarchies:*
   An essential step when approaching the music classification field is to understand the many issues related to how culture and sub-groups define musical categories, construct taxonomies and form interrelationships between categories. These issues have been discussed in the work of Fabbri [40], Brackett [19], Pachet and Cazaly [67], Aucouturier and Pachet [11], just to mention a few. From a data warehouse point of view, the taxonomies presented shared common properties. In a multidimensional database, a dimension hierarchy is said to be: *strict*, if all dimension values have no more than one direct parent, *onto*, if the hierarchy is balanced, and *covering*, if no containment path skips a level [71]. It is clear that, e.g., in a genre dimension, the hierarchy would be non-strict, non-onto and non-covering. However, this is not sufficient. Since very little consensus exists between taxonomies, the techniques already existing for slowly changing dimensions in multidimensional databases may not be appropriate. Instead, MWs require support for versioning abilities, mimicking software versioning systems such as CVS[5] or Subversion[6], where different hierarchies could coexist and evolve.

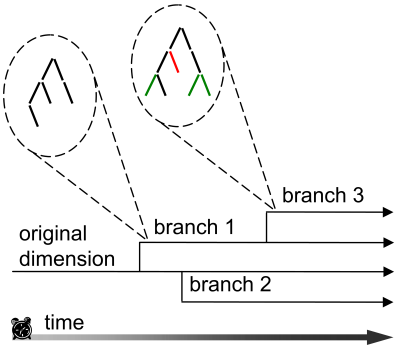


Figure 2.9: Versioned Hierarchies

   Figure 2.9 shows a versioned genre hierarchy that, for example, defines a classification of the genre dimension for different user profiles. Versioned hierarchies call for the creation of new database operators. Examples of such operators are: navigation in the different branches of the hierarchy, comparison of the branches between users, evolution of a user hierarchy over time, and ability to derive branches from existing ones.

7. *Fuzzy hierarchies:*
   Non-strict hierarchies, i.e., hierarchies supporting elements having multiple

[5] http://www.nongnu.org/cvs/
[6] http://subversion.tigris.org/

parents, allow different paths to be followed when performing roll-up operations. In the time dimension, days can be rolled-up into months and in turn into years. Similarly, days can be rolled-up into weeks by following a different path since there are overlaps between week–month and week–year precision levels.

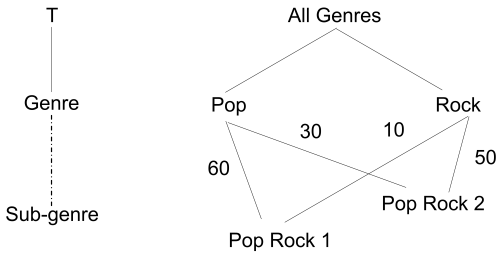


Figure 2.10: Fuzzy Hierarchy of the Genre Dimension

The expected contribution of the project will consist in the creation of a data model and its algebra enabling the use of *fuzzy hierarchies*. Fuzzy hierarchies enable "children" to belong to multiple parents with various degrees of affiliation evaluated through membership functions as defined by fuzzy logic. While fuzzy hierarchies are not required to handle typical data warehousing demands, they become unavoidable for MWs in order to represent complex hierarchies such as in the genre dimension as illustrated in Figure 2.10. Sub-genres would belong to genres to a certain degree, e.g., the genre Jazz-Rock could belong 60% to Jazz and 40% to Rock, a notion that multidimensional data models have not been able to fully capture so far.

8. *Navigation in n-dimensional space:*
   The mental representation of songs as objects in an n-dimensional space is not new in the field of music classification. Far from being purely a dream, projects such as MusicMiner [7] already offer a two-dimensional mapping of personal music collections.

   It is therefore very tempting to enrich the MW data model with multidimensional navigation features such as notions of neighborhood, intersections, landscape, fuzzy borders, etc. In such a space, a play-list can be seen as a journey from one song to another. Automatic play-list generation could be as like car navigation systems able to recommend some itineraries with notions of primary and secondary roads to reflect the musical tastes of the user.

9. *Aggregates for dimensional reduction:*
   A very challenging aspect of MWs is the high number of dimensions used,

[7] http://musicminer.sourceforge.net/

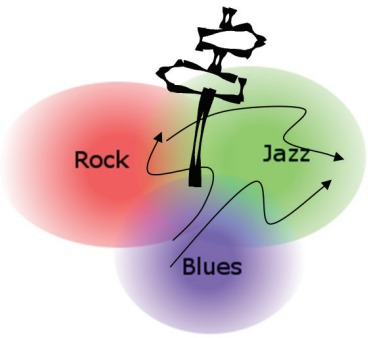


Figure 2.11: Navigation in the Musical Space

songs can be described using several hundred dimensions, hence, urging the need for efficient ways to aggregate this massive amount of information in useful ways. The traditional multidimensional approach is to reduce dimensionality by using projection, i.e., throwing out dimensions by omitting pieces of available information. Instead, by using *fused dimensions*, many dimensions, such as the rhythm, the notes, and the loudness could be summarized into a more general one, reflecting the overall melody of the songs. Using aggregates for dimensional reduction clearly offers many advantages as the complexity of the data is reduced, while the essence is maintained. The MW should provide efficient techniques to reduce or increase the number of dimensions.

10. *Integration of new data types:*
The musical world does not only deal with audio but also embraces a lot of external multimedia content. New bands often aim to increase their audience by creating interactive web sites, video clips, attractive CD covers, etc. MWs should be able to deal with such information, as not including these non-audio additions is neglecting an increasingly important part of the musical experience of the audience. Interactive web sites, for example, often make extensive use of embedded application such as Flash. These applications offer biographies, an agenda of next concerts, rumors and forums to users. MWs should provide users such pieces of information. It should be possible to define specific extractors for the applications and to perform analysis on the extracted features. MWs should be able to handle queries requiring partial integration of the applications, e.g., obtaining the list of Madonna's next concerts.

2.7 Conclusions and Future Work

Inspired by the previous successes of DWs in business integration issues and on-the-fly analytical demands, this chapter proposes the development of MWs which are centralized data stores based on the data warehousing approach and optimized to answer the fast-search needs of future large music information retrieval systems.

Previous work on musical classification has shown multiple sources of inconsistencies between ontologies. One source of these inconsistencies is the use of different musical facets when describing a song. These facets can be described using musical metadata. Four high-level categories of metadata are identified and briefly described: editorial, cultural, acoustic and physical metadata. Using these four categories, a case study of musical database management is presented. Based on the case study, we propose an approach to develop a on-line music recommendation system able to find the song the most similar to another with respect to one or a combination of musical aspects and to manipulate music playlists.

Finally, ten exciting challenges offered by MWs for the existing DWs are identified. While these challenges originate from the requirements music classification systems, they are, however, not confined to this area. In particular, data imperfections, precision-aware retrieval using coarse Top-K queries or streams, versioned irregular hierarchies and fuzzy hierarchies are new and relevant to the general database research community.

Work on these challenges should later be pursued in order to support the successful integration of DWs in the musical world.

Chapter 3

High-Level Audio Features: Distributed Extraction and Similarity Search

Today, automatic extraction of *high-level* audio features suffers from two main scalability issues. First, the extraction algorithms are very demanding in terms of memory and computation resources. Second, copyright laws prevent the audio files to be shared among computers, limiting the use of existing distributed computation frameworks and reducing the transparency of the methods evaluation process. The *iSound Music Warehouse* (iSoundMW), presented in this chapter, is a framework to collect and query high-level audio features. It performs the feature extraction in a two-step process that allows distributed computations while respecting copyright laws. Using public computers, the extraction can be performed on large scale music collections. However, to be truly valuable, data management tools to search among the extracted features are needed. The iSoundMW enables similarity search among the collected high-level features and demonstrates its flexibility and efficiency by using a weighted combination of high-level features and constraints while showing good search performance results.

3.1 Introduction

Due to the proliferation of music on the Internet, many web portals proposing music recommendations have appeared. As of today, the recommendations they offer remain very limited: manual tagging has proved to be time consuming and often

results in incompleteness, inaccuracy and inconsistency; automatic tagging systems based on web scanning or relying on millions of users are troubled, e.g., by mindless tag copying practices, thus blowing bag tags. Automatic extraction of music information is a very active topic addressed by the Music Information Retrieval (MIR) research community. Each year, the Music Information Retrieval Evaluation eXchange (MIREX) gives to researchers an opportunity to evaluate and compare new music extraction methods [33]. However, the MIREX evaluation process has proved to be resource consuming and slow despites attempts to address these scalability issues [20,37]. So far, concerns with copyright issues have refrained the community to distribute the extraction among public computers as most algorithms require the audio material to be available in order to perform the feature extraction [1]. The features are therefore extracted from a relatively small music collection that narrows their generality and usefulness. Additionally, the feature extraction, being run by private computers on a private music collection, limits the transparency of the evaluation process. These limitations call for the development of a system able to extract meaningful, high-level audio features over large music collections. Such a system faces data management challenges. Noteworthily, the impressive amount of information generated requires an adapted search infrastructure to become truly valuable.

Our intention is to create a system able to cater for different types of features. Present literature mainly focuses on features that have either absolute or relative values, thus motivating the handling of both kinds of features. In this paper, the exact selection of the features is actually not as important as it is to demonstrate how extraction can be handled on public computers and enabling researchers to compare results obtained by using different algorithms and features.

The contributions of this chapter are two-fold. First, we propose a framework for collecting high-level audio features (that were recently proposed by [22,23,24,50]) over a large music collection of 41,446 songs. This is done by outsourcing the data extraction to remote client in a two-step feature extraction process: (1) dividing the audio information into short term segments of equal length and distributing them to various clients; and (2) sending the segment-based features gathered during step one to various clients to compute high-level features for the whole piece of music. Second, we propose a flexible and efficient similarity search approach, which uses a weighted combination of high-level features, to enable high-level queries (such as finding songs with a similar happy mood, or finding songs with a similar fast tempo). Additionally, to support the practical benefits of these contributions, we propose a short scenario illustrating the feature extraction and search abilities of the iSoundMW.

For the general public, the iSoundMW music offers recommendation without suffering from a “cold start”, i.e., new artists avoid the penalties of not being well

[1] https://mail.lis.uiuc.edu/pipermail/evalfest/2008-May/000765.html

known, and new listeners obtain good music recommendation before being profiled. For researchers, the iSoundMW (1) offers flexible search abilities; (2) enables visual comparison of both segment-based and aggregated high-level features; (3) provides a framework for large scale computations of features; and (4) gives good search performances.

The remainder of this chapter is organized as follows. Related work is presented in Section 3.2. Section 3.3 offers an overview of the system, explains the process of collecting the high-level audio features, describes how similarity search with weighted coefficients is performed, and how the search can be further optimized by using range searches. Section 3.4 illustrates the similarity search on a concrete example. Section 3.5 concludes and presents future system improvements and research directions.

3.2 Related Work

Research on distributed computing has received a lot of attention in diverse research communities. The Berkeley Open Infrastructure for Network Computing framework (BOINC) is a well-known middleware system in which the general public volunteers processing and storage resources to computing projects [4, 5] such as SETI@home [6]. However, in its current state, BOINC does not address copyright issues, does not feature flexible similarity search, and does not enable multiple steps processes, i.e., acquired results serve as input for other tasks. Closer to the MIR community, the On-demand Metadata Extraction Network system (OMEN) [62] distributes the feature extraction among *trusted nodes* rather than public computers. Furthermore, OMEN does not store the computed results, and, like BOINC, does not allow similarity search to be performed on the extracted features.

Audio similarity search is often supported by creating indexes [36]. Existing indexing techniques can be applied to index high dimensional musical feature representations. However, as a consequence of the subjective nature of musical perception, the triangular inequality property of the metric space is typically not preserved for similarity measures [58, 74]. Work on indexes for non-metric space is presented in the literature [43, 80]. Although the similarity function is non-metric, it remains confined in a pair of lower and upper bounds specifically constructed. Therefore, using these indexes would impose restrictions on the similarity values that would limit the flexibility of the iSoundMW.

3.3 System Description

In this section, we present an overview of the system followed by a more detailed description of a two-step process for extracting high-level features. Later, we describe

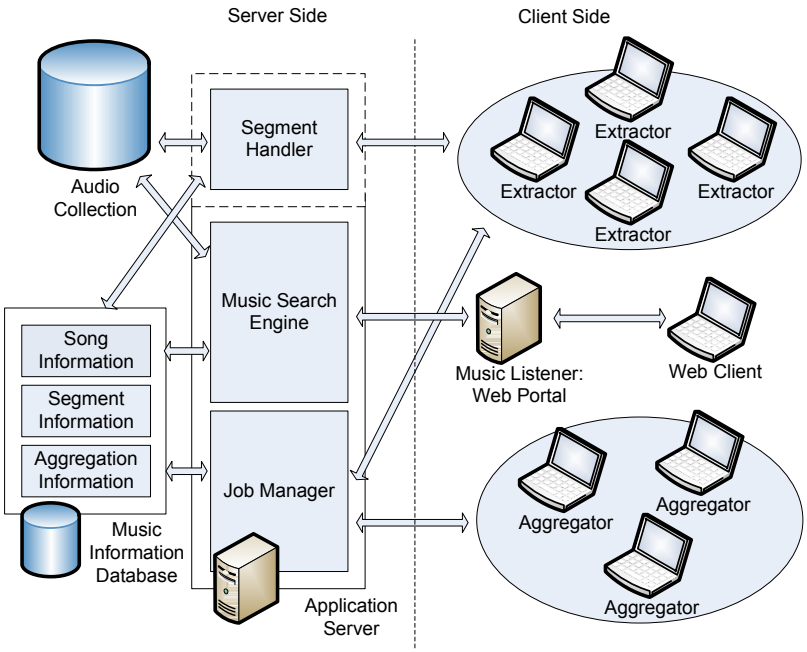


Figure 3.1: System Architecture

how flexible similarity searches are performed and how they are further optimized using user-specified constraints.

3.3.1 iSoundMW Overview

The iSoundMW system has a client-server architecture, shown in Figure 3.1. The server side is composed of a central data repository and an application server that controls the extraction and similarity searches. The data repository stores all the audio collection in MP3 format and uniquely identifies each by a number. It also contains all the editorial information, e.g., the artist name, the album name, the band name, the song title, the year, the genre, and copyright license, and the physical information, e.g., the file size, the format, the bit rate, and the song length, that are stored in the music information database. Additionally, the Music Information Database holds all the extracted feature information.

The application server is composed of three distinct components. First, the *job manager* assigns extraction or aggregation tasks to the clients, collects the results, and prepares progress reports about each extraction process. Second, the *segment handler* splits the audio information into short term segments overlapping or non-overlapping of equal length, and makes them available to the clients they have been assigned.

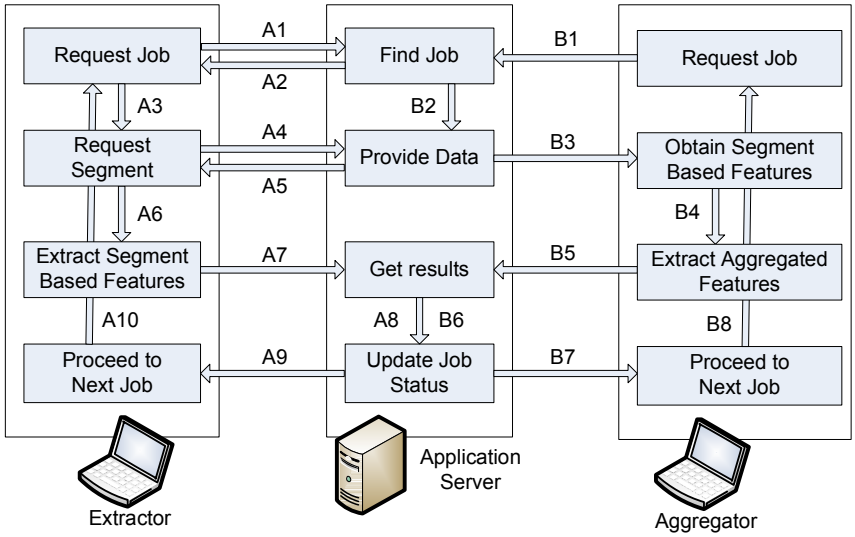


Figure 3.2: Two-step Extraction Process

Future versions of the system will have multiple segment handlers, enabling multiple music collections to be analyzed without violating copyright for any of them. Third, the *music search engine* serves requests such as finding the most similar song to a given seed song with respect to a combination of features.

The client side is composed of three different types of clients. First, the *segment extractors* are receiving short term segments, and extracting their high-level features, e.g., the pitch, or the tempo. Second, the *segment aggregators* are performing computations based on the high-level features of the segments to produce other high-level features requiring features over different segments, e.g., the mood. Third, the *music listeners* perform similarity search queries on the extracted high-level features.

3.3.2 Collecting the High-Level Features

The increasingly fast growth of music collections motivates the adoption of a distributed approach to perform feature extraction. Such approach, however, brings forward copyright issues, i.e., the copyrighted material in the music collection prevents the audio content to be freely distributed. We propose to address this issue by performing the feature extraction in a two-step process. In the proposed two-step extraction process, the full songs are not available to, or re-constructible by, the computers performing the extraction of the audio features.

Step 1, as illustrated by the A arrows in Figure 3.2, consists of dividing the audio information into short term segments of equal length and distributing them to the clients. Dividing the songs into segments offers the following advantages. First,

it limits and normalizes the memory size and processor time needed on the client side to perform the feature extraction, as most extraction methods require resources proportional to the length of the audio source. Second, it avoids copyright issues as the audio segments are very short and distributing them falls under "fair use" since the segments are only temporally stored in memory on the client, and the full songs cannot be reconstructed by the clients.

Step 2, as illustrated by the B arrows in Figure 3.2, consists of sending the segment-based features gathered during step one to the clients. Using the features of step 1, the clients are computing high-level features for the whole piece of music. The computation of the aggregated features can be performed over the features obtained from multiple segments, as they are not subject to copyright issues. While having the segment based and the aggregated high-level features computed separately represents a potential overhead, it remains insignificant compared to the resources needed to perform each of the two feature extraction steps.

A *job* is the smallest atomic extraction or aggregation task that a remote client has to perform. Each job is uniquely identified for each extraction method and is composed of the following attributes: a song reference, a segment number, a starting point, an ending point, a lease time, a client reference, a counter of assignments, and a result holder. The assignment of jobs to clients is a critical part of the segment extraction process. Some randomness in the jobs assignment prevents clients to reconstruct the full song from the distributed segments as the segments assigned to a client will belong to different songs. However, since all the segments of a song have to be processed by step 1 before moving to step 2, assigning the segments randomly to clients delays the obtainment of results. Some locality constraints are therefore enforced in order to quickly acquire preliminary results and proceed to step 2.

In order to control which job should be assigned next, jobs are assigned in sequence. To avoid all the clients trying to obtain the same job, the assignment of a job is relaxed from being the minimal sequence number to being one of the lowest numbers in the sequence. Assigning jobs "nearly" in sequence still allows the application server to control which jobs should be prioritized for segment extraction and feature aggregation, e.g., when a similarity search is requested for a new song without any features extracted.

The current configuration of the system has shown its usability on a music collection of 41,446 songs composed of 2,283,595 non-overlapping 5 seconds segments. In terms of scalability, the job manager, supported by a PostgreSQL 8.3 database running on an Intel Core2 CPU 6700 @ 2.66GHz, 4GB of RAM under FreeBSD 7.0, was able to handle 1,766 job requests and submissions per second. Running on the same computer, the job manager and the segment handler were able to serve 87 clients per second; the bottleneck being the CPU consumption mostly due to the on-the-fly segmentation of the MP3 files. At this rate, an average bandwidth of 13,282 KB/s

Abssim Table

SongID	Tempo	Pitch
1	1	0
2	2	3
3	3	2
...	...	...

Relsim Table

SeedID	SongID	Timbre
1	1	0
1	2	6
1	3	2
...	...	...
2	1	6
2	2	0
...	...	...

Figure 3.3: The Absolute and Relative Similarity Tables

was consumed to transfer the segmented files of the data collection. Given that, by experience, the average processing of a segment by a modern desktop computer takes 5 seconds, the presented configuration would be able to handle over 400 clients. In a setup where the segment handler is run separately and not considering network limitations, the job manager could serve close to 9000 clients.

3.3.3 Similarity Search

Search among the collected features is a valuable tool to compare and evaluate extraction results. Similarity search raises two main challenges. First, similarities are of two types: similarities that can be computed rapidly on-the-fly and similarities that have to be pre-computed. Second, each similarity is tweaked dynamically with different user defined weight coefficients in order to adjust the final similarity value. In the following, we propose to find the 10 songs the most similar, with respect to a user defined weighted combination of features, to a given seed song.

Similarities that can be computed on the fly are stored in a single table, *abssim*: (“songID”, “abs1”, “abs2”, ...), where the songID is the primary key identifying the song and abs1, abs2, ..., are the different attributes. The table is composed of 41,446 songs and 18 attributes, such as the tempo, the motion, the articulation, the pitch, and the harmonic. Similarities that cannot be computed on the fly have to be pre-computed and stored for each pair of songs as some similarities are not symmetric and do not fulfill the triangular inequality. They are stored in a second table, *relsim*: (“seedID”,“songID”,“rel1”), where the “seedID” and “songID” refers to a pair of songs and “rel1” is the similarities value between the two songs. The relsim table has 1.7 billion rows and a timbre attribute. The abssim and relsim tables are illustrated in Figure 3.3.

The distance function, the similarity search is based on, is as follows:

$$\begin{aligned}\text{dist}(x,y) = a \times |\ \text{pitch}(x,y)| + b \times |\ \text{tempo}(x,y)| \\ + c \times |\ \text{timbre}(x,y)| + \ldots\end{aligned}$$

where x and y are two songs, tempo, pitch, and timbre are the computed differences between x and y for each feature, and a, b, and c are their respective coefficients. The pitch difference, like the tempo, can be computed on the fly: $\text{pitch}(x,y) = x.\text{pitch} - y.\text{pitch}$. The timbre difference is pre-computed and requires a lookup in the *relsim* table.

Assume the following query: find the 10 songs with the lowest distance from a given seed song. First, we compute the difference between the values of the seed song and the values of all the other songs in the table "abssim". This requires a random disk access using an index on the "songID" to read the values of the seed song, followed by a sequential scan to compute on the fly all the similarity values from the "abssim" table. Second, using an index on the "seedID", we select all the pre-computed similarities that correspond to the query songs. The song pairs with an identical "seedID" are contiguous on disk to minimize the number of disk accesses. Third, the 41,446 similarities from both resulting sets have to be joined. Fourth, the final distance function is computed and the 10 closest songs are returned.

Hash join	123 ms
Merge join	141 ms
Nested loop	163 ms
Top K	405 ms
Total runtime	575 ms

Table 3.1: Time Cost of Similarity Search

Table 3.1 shows the average query processing time of the most costly operations involved in queries run on the MW described in Section 3.3. The performance is acceptable; most of the query processing time is cause by the join (20%) and the ordering (75%) operations. Hash joins have shown slightly better performance than, merge sort and nested loop joins. Merge joins should provide the performance results if the sort operation can be avoided by respecting the data organization on disk.

3.3.4 Similarity Search within a Range

The query cost is mainly due to the join and the Top-K operations over the whole music set. We introduce similarity searches within ranges to reduce the size of the set on which the join and the Top-K are performed, thus decreasing the search time. We

propose to search, within a user specified range for each feature, for the most similar songs to a given seed song.

The abssim table is used for each similarity search. Its small size allows sequential scans to be performed fast. Therefore, filtering out the values outside the query range effectively reduces the search time. Similarly, performing a filtering of the selected rows from relsim contributes to reducing the search time. However, several additional improvements can be made, they are illustrated in Figure 3.4.

First, a partial index on the seedID for the similarity values that are below a given threshold can be created. If the partial index has a good selectivity, speed is gained by trading the sequential scan for a few random disk access. This is of critical importance as the database grows larger. Using a threshold with a too high selectivity reduces the chances of the partial index being used.

Second, to further improve the search, one can create arrays containing pairs of songID and similarity value for each seedID. Arrays offer the following advantages. As the partial index, only the similar values are accessed, thus greatly reducing the cost of filtering. The similar values are clustered on disk for each seed song, similarity values are stored in the array in ascending order, and the array itself is compressed, therefore allowing a complete array to be retrieved in a single disk access. Accessing

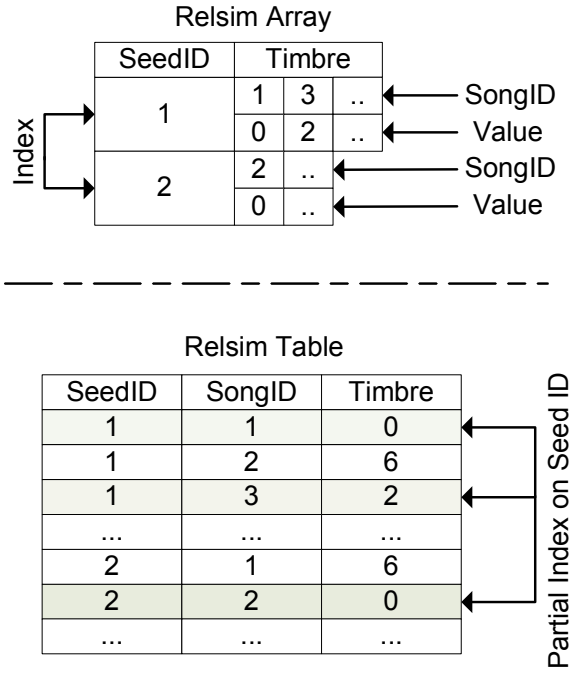


Figure 3.4: Range Queries with Arrays or a Partial Index

	Table	Array	Table + PI
Query time (ms)	130	15	54
Query time (%)	100	12	42
Size (MB)	70000	71550	70200
Size (%)	100	112	100

Table 3.2: Cost of Similarity Search within a Range

similarity values is done in an efficient way: one index lookup for locating the array, and one disk access to read the pairs of songID and value.

Table 3.2 presents the costs in search time and in disk space of each approach for the same set of constraints. As expected, specifying ranges for the similarity search significantly improves the response time compared to searching in the complete music collection; a simple filtering can improve the response time by a factor of 4. Using a partial index or arrays, further decreases the search time but come with a storage cost [30]. The search time improvements are dependent on the selected range as well as the selectivity of the threshold chosen for both the array and the partial index.

Similarity searches using the partial index are slower than arrays due to the random disk accesses that are required to read the values. However the partial index offers two major advantages. First, it does not require any backstage processing; the partial index is updated as new data is entered in the relsim table. Updating the arrays requires some additional processing that could be performed automatically with triggers. Second, the choice of using the partial index or not is delegated to the query planner, all queries are treated transparently regardless of the range chosen, i.e., no extra manipulation is needed to handle the array organization of the data when a threshold is reached.

3.4 The iSoundMW

A web interface is connected to the iSoundMW. It offers a visual understanding of the functioning of the system and gives an opportunity to observe the information extracted from different songs and compare the result with the music being played simultaneously.

The setup is as follows. The music collection has a size of 41,446 songs in MP3 format, segments are non-overlapping and have a 5 seconds length. The Music Information Database is partially loaded with some segments information and some aggregated feature information, but some segments still have to be extracted and aggregated. Some clients are connected to the application server and are processing some segment extraction and feature aggregation jobs. If all the segments have not

been extracted, the corresponding jobs will be prioritized. to ensure that the extracted features are available. The user interface, as shown in Figure 3.5, consists of a music player and a graph showing the content of the extraction as the music is being played.

The main steps of a short scenario are presented below.

Step 1 A user searches for a famous song and provides its title, artist name, or album name, e.g., the user enters "Madonna" for the artist and "Like" for the title. The system retrieves a list of maximum 20 candidates present in the database based on the ID3 tags of the MP3 in the music collection, 3 songs in this scenario. The song "Like a Prayer" is listed twice, as it belongs to two different albums. The user selects one of the two "Like a Prayer" songs.

Step 2 The system searches for the 10 most similar songs to the song selected, places them in the playlist, and starts playing the songs. The most similar song to the song selected is generally the song itself, therefore the song appears first in the generated playlist. At this stage, the search is based on default coefficients and one the complete database. The second version of the song "Like a Prayer" appears further in the generated playlist.

Step 3 As the song is being played, the tempo analysis based on the extracted segments and the aggregated features is displayed in the graph. For each 5 seconds (corresponding to a segment length), new points are placed on the graph. If the segments have not been extracted, a request is sent to the application server to prioritize the corresponding jobs. The graph updates as new extraction results are arriving from the extraction and aggregation clients.

Step 4 Any extracted features given on an absolute scale can be displayed on the graph, e.g., the user has selected to display the mood features.

Step 5 Moving to the playlist configuration panel, the user can select, for each of the extracted features, the weight to be used as a coefficient for the similarity search, e.g., we choose to put a high coefficient on the pitch and the mood. Once the tuning of the weights is done, when the user selects a new song, the system searches for the songs that, with the given coefficients, are the most similar to the song currently being played. Additionally, the user can select a range of values in which the search should be performed.

Step 6 When similar songs are being played, the user can select to compare the currently played song with the song originally selected to generate the playlist, e.g., the main difference between the two versions of "Like a Prayer" rely in the harmonic feature.

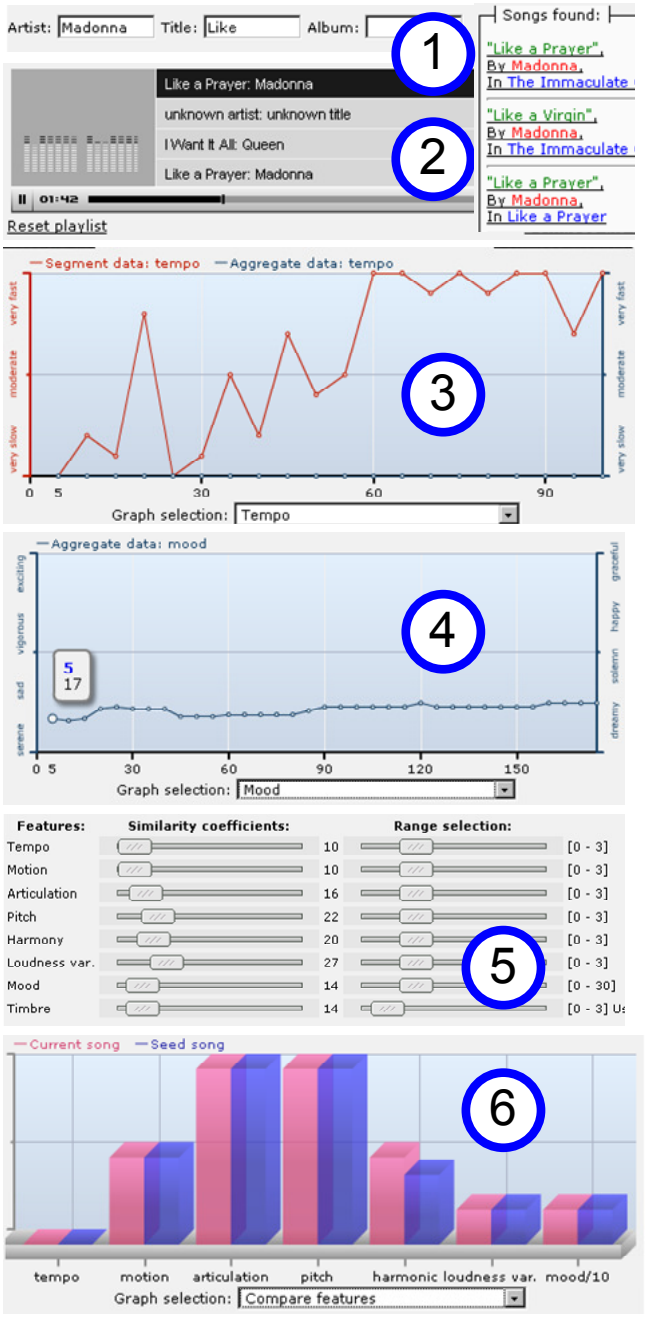


Figure 3.5: Web Interface

3.5 Conclusions and Future Work

The automatic extraction of high-level features over a large music collection suffers from two main scalability issues: high computation needs to extract the features, and copyright restrictions limiting the distribution of the audio content. This chapter introduces the iSoundMW, a framework for extracting high-level audio features that addresses these scalability issues by decomposing the extraction into a two-step process. The iSoundMW has successfully demonstrated its ability to efficiently extract high-level features on a music collection of 41,446 songs. Furthermore, the iSoundMW proves to be efficient and flexible for performing similarity searches using the extracted features. This is done by allowing users to constrain the search within a range and specify a weighted combination of high-level features. To further optimize the search, three different approaches are compared in terms of query time and storage. The threshold for building the partial index and arrays are decisive parameters to obtain good search performance.

Future work encompasses integrating different similarity functions in the features search, providing comparison between them, and enabling user feedback and its reuse for user specific recommendation.

Appendix

3.A Son of Blinkie

The present section introduces Son of Blinkie (SoB), an interactive application for audio extraction and music classification. SoB results from a 3 months long collaboration with the International Music Information Retrieval Systems Evaluation Laboratory (IMIRSEL) at the University of Illinois at Urbana-Champaign.

Our personal contributions include the design of the new system architecture, the definition of the API, and the development of the new system components. The latest version of SoB obtained the second place at the JCDL'08 demo contest.

A major component of SoB is the dataflow platform referred to as Meandre. In this Appendix, we propose a short introduction to Meandre, and present SoB and its evolution. Finally, we draw a brief comparison between SoB and the iSoundMW and offer future work directions.

3.A.1 Meandre

Meandre is defined by its authors as "a semantic enabled web-driven, dataflow execution environment" [1]. It provides a framework for assembling and executing applications based on data flows from already existing components. In a *data flow application*, each software components processes data, e.g., it accesses a data store, transforms data, performs analyses, or offers a visualization of the results. Within Meandre, each flow is represented as a graph that shows executable components (i.e., basic computational units, or building blocks) as icons linked through their input and output connections. Based on the inputs and properties of a executable component, a unique output is generated upon execution. Meandre also provides publishing capabilities for flows and components, enabling users to assemble a repository of components for reuse and sharing. This allows users to leverage other research and development efforts by querying and integrating component descriptions that have been published previously at other shareable repository locations.

As shown in Figure 3.6, Meandre is originally composed of 3 subsystems, namely Meandre Core, Meandre Manager, and Meandre Workbench. They are described below.

Meandre Core Dataflow execution is based on the idea of applying transformational operations to a flow or stream of data. In a data-driven model, data availability determines in what sequence code instructions are executed. Meandre Core is the subsystem responsible for the execution of the flows.

Meandre Manager Meandre Manager is a web application handling user and job management administration for the Meandre web services-based engine. In

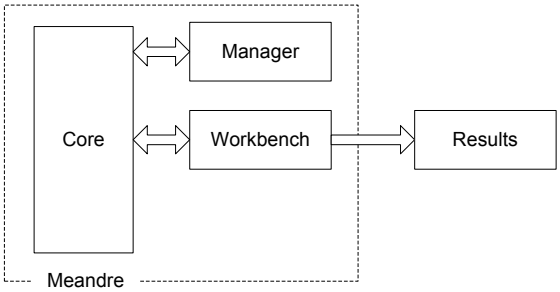


Figure 3.6: Overview of Meandre Components

addition, it provides an interface to allow a user to create a code template for new components as well as the ability to add new components. The Meandre Manager allows for the display of components and flows, and the management of the flows in execution.

Meandre Workbench Meandre Workbench is a visual programming environment that allows users to easily connect software components together in a unique data flow environment. The Workbench can be used to develop diagrams of data operations to be performed. Each operation is represented by an icon, and the icons are linked together in a flow representing the movement of data through each operation. Each of these icons represent a component. The software components are reusable components that facilitate collaboration among developers. They can be written in Java, Python, or Lisp. Components and flows have tags and additional meta data associated with them that can be used to assist in searching and sorting.

3.A.2 Development of Son of Blinkie

Son of Blinkie (SoB) is an interactive web application allowing researchers to compare classification models on public music collections. SoB is operated on top of Meandre's service-oriented architecture, taking the form of a series of reusable, open-source components managed by and executed as a shareable workflow. Not only can users run SoB against their own data sets with Meandre, but they can also reuse and modify components and workflows to build their own music research applications, such as classification tools.

Each component represents one step in processing the data. The components run in the order established by the flow: from receiving the song filename and model filenames from the web application, to loading the audio and model data into memory, to extracting a variety of features from the song, to applying the model to the

extracted features, and to returning the predicted results to the web application for visualization. Every time a different song is selected, the web application executes this same flow.

3.A.2.1 SoB 1

Version 1 of SoB is relying on a new output component that was generating the HTML code for the web interface. The advantage of this approach was to avoid heavy modifications of the architecture of Meandre. However, it also presented some inconveniences; the users had to execute workflows by accessing Meandre Manager, then connect to the flow and wait for the workflow to generate an HTML output.

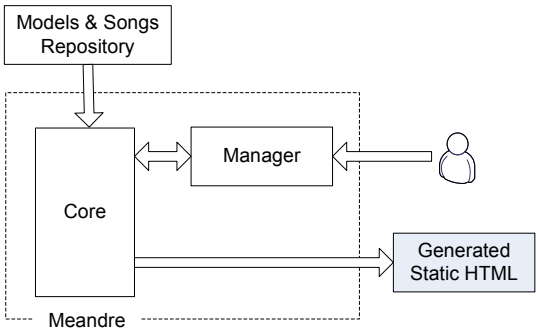


Figure 3.7: Architecture of SoB 1

The web application is totally dependent of the new HTML generation component, thus complicating its development and the management of user interactions.

3.A.2.2 SoB 2

Version 2 of SoB uses an external web application and Meandre as a data provider. The advantage is to provide an interactive web interface. The user can now select which models and songs should be analyzed. Results are shown in synchronization with the music. It is also possible to browse forward and back in the results by selecting a position on the audio track.

As shown in Figure 3.8, the architecture of Meandre had to be modified to allow applications to initiate dataflow executions and obtain results back. These modifications resulted in the development of an API. Such architecture, however, still presents drawbacks for a web application. A part of the application logic is now executed on the client side through JavaScript. While the API was functional, it is not fully mature, e.g., obtaining a flow URL requires a listing existing flows, creating a new flow,

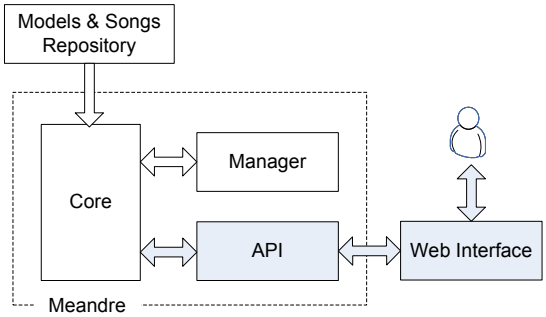


Figure 3.8: Architecture of SoB 2

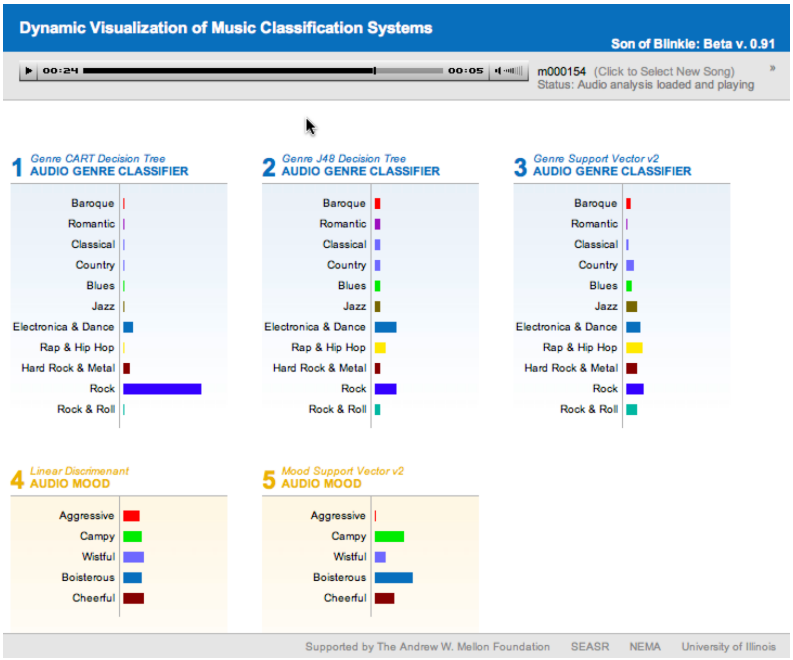


Figure 3.9: User Interface

listing the existing flows, and identifying the newly created flow. A screenshot of the interface is provided in Figure 3.9.

The data output from the API is XML. On long songs and when selecting many classification models, the XML output tends to be very voluminous. We experienced that XML parsers (Sax) use a lot of memory and cause the web browser to freeze. A more usable data representation should therefore be used.

3.A.2.3 SoB 3

Version 3 of SoB is still under development. It relies on a proxy server between Meandre's API and the user application. The proxy quickly transforms the results in XML format from Meandre into JavaScript Object Notation (JSON) that is directly usable by the web application. Using JSON eliminates the need of parsing XML results. The JSON encoded values can be directly interpreted in JavaScript, thus allowing long results to be handled by the web application. Another main advantage of the proxy is the possibility to act as a cache, avoiding multiple identical requests to be performed. The caching is done with a PostgreSQL database.

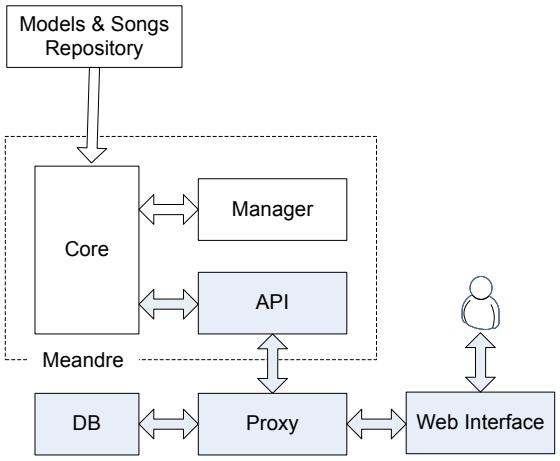


Figure 3.10: Architecture of SoB 3

3.A.3 Future Developments

ISoundMW and SoB 3 perform similar tasks in very different ways. Future work is to develop an application able to take the best of each system. A brief comparison between the ISoundMW and SoB 3 is provided below, it is followed by an explanation on how the two systems could be merged.

Comparison between ISoundMW and SoB	
ISoundMW works with copyrighted materials, the songs do not have to be publicly available.	SoB requires songs to be fully accessible by Meandre.
ISoundMW does not offer real-time computation of the features but is based on batch processing with priority queues.	Sob is executing the work flow on demand.
ISoundMW does not allow algorithms to be changed once distributed to the clients.	SoB allows the algorithms to be changed at any time and allows power users to define new models.
ISoundMW performance scales to a large number of clients and users.	SoB performances are only acceptable when few users are connected (< 10).
ISoundMW offers post-extraction processes involving multiple songs, e.g., song similarity search.	SoB does not support any post-processing of current or already computed results.

We propose to integrate the two systems as presented in Figure 3.11. An automatic updating system allows to update client extraction and aggregation algorithms. This system thus interacts with the ISoundMW, the extraction model repository, and the computing cloud. The computing cloud remains controlled by the ISoundMW. Meandre acts as a special node of the computing cloud. As other nodes, it receives updates from the automatic updating system and flow execution requests, i.e., jobs, from the ISoundMW. Meandre differs from other nodes as it can access, if necessary, the total length of the audio data to be extracted. The ISoundMW job assignment system is improved to allow the assignment of specific jobs to particular clients. High priority jobs are thus assigned to Meandre to ensure fast analysis. The extracted data is indexed by the ISoundMW to allow efficient search among the features and ensure reusability of previously completed extraction tasks. The database component of the ISoundMW will allow the system to deal with thousands of users. The web interface directly interacts with the ISoundMW.

This setup takes the best of ISoundMW and SoB. The integrated system would be able to act on both public and private music audio sets. The system would be

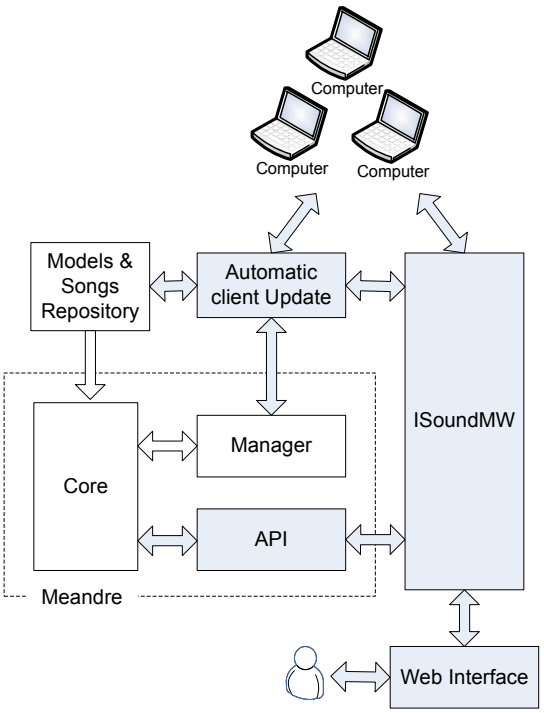


Figure 3.11: Integration of IsoundMW and SoB

able to offer real-time computation for a restricted number of high priority jobs. For jobs that do not require immediate results, the system is scalable to gigantic music collections by using the computational cloud. Thanks to the automatic client update system, the computational nodes are running updated extraction and analysis models. Finally, the system is able to re-use results from previously computed jobs.

Chapter 4

Using Fuzzy Song Sets in Music Warehouses

The emergence of music recommendation systems calls for the development of new data management technologies able to query vast music collections. In this chapter, we present a *music warehouse* prototype able to perform efficient nearest neighbor searches in an arbitrary song similarity space. Using fuzzy songs sets, the music warehouse offers a practical solution to three concrete musical data management scenarios: *user musical preferences*, *user feedback*, and *song similarities*. We investigate three practical approaches to tackle the storage issues of fuzzy song sets: *tables*, *arrays*, and *compressed bitmaps*. We confront theoretical estimates with practical implementation results and prove that, from a storage point of view, arrays and compressed bitmaps are both effective data structure solutions. With respect to speed, we show that operations on compressed bitmap offer a significant grain in performances for fuzzy song sets comprising a large number of songs. Finally, we argue that the presented results are not limited to music recommendations system but can be applied to other domains.

4.1 Introduction

Automatic music recommendation systems have recently gained a tremendous popularity. To provide pertinent recommendations, music recommendation systems use fuzzy set theory [98] to combine user profiles, music features, and user feedback information. However, at the current growing speed, the database element of any recommendation system will soon become a bottleneck. Hence, appropriate musical

data management tools, able to manipulate fuzzy sets and scale to large music collection and growing user communities, are needed. Music Warehouses (MWs) are dedicated data warehouses optimized for the storage and analysis of music content.

The contributions of this chapter are fourfold. First, based on a previous case study [29], we propose three generic usage scenarios illustrating the current demands in musical data management. To answer these demands, we define fuzzy song sets and develop a query algebra for them. Second, to demonstrate the usefulness of fuzzy song sets, a prototypical MW composed of two multidimensional cubes is presented. Fuzzy song sets prove to be an adequate data representation to manipulate musical information. Third, we discuss three solutions for storing fuzzy song sets and fuzzy sets in general. We construct theoretical estimates for each storage solution. A practical implementation shows that the storage overhead represents a major part of the storage consumption and that two solutions are viable for large music collections. Fourth, we benchmark and compare the performance of the main operators previously presented for various sizes of both data structures. Experiments are conducted on a real music collection.

This chapter demonstrates how fuzzy set theory can be used in the context of music recommendation systems. All results presented in this chapter can be directly applied to standard fuzzy sets; the presented storage solutions remain generic and can thus be applied to a vast range of domains besides music recommendation and user preferences.

The remainder of this chapter is organized as follows. After presenting related work on fuzzy sets for the management of musical data in Section 4.2, we present three information scenarios that are commonly treated by music recommendation systems in Section 4.3. We proceed in Section 4.4 by defining fuzzy song sets and an algebra. In Section 4.5, two prototypical multidimensional cubes are presented; they illustrate the use of the algebra through queries examples. Storage solutions are then discussed in Section 4.6. For each, precise storage estimates are proposed and experimentally validated. Next, in Section 4.7 a comparison of the performance of the fuzzy song set operators on the bitmap and array representations is conducted. Finally, we conclude and describe promising future research directions in Section 4.9.

4.2 Related Work

Research on music recommendation systems has received a lot of attention lately. Current trends on playlist generation are focused on how to improve recommendations based on user-specific constrains. For example, a playlist generator that learns music preferences by taking user feedback into account was presented by [70]. Other new interesting approaches concentrate on aggregating different music features; for instance in [18], the use of generalized conjunctions and disjunctions of

fuzzy sets theory for combining audio similarity measures is studied. However, fewer researchers have addressed the scalability issues raised by these methods in terms of storage and performance [12, 68]. This chapter focuses specifically on the storage and performance issues and proposes to manipulate a large collection of musical data where song similarities, user preferences and user feedbacks are represented with fuzzy sets.

A traditional database approach is to use a relational model such as the one proposed by Rubenstein that extends the entity-relationship data model to implement the notion of hierarchical ordering, commonly found in musical data [79]. A multimedia data model, following the layered model paradigm that consists of a data definition layer, a data manipulation layer, a data presentation layer, and a control layer, is presented in [95], but no query language is proposed. None of those models adopts a multidimensional approach by representing data in cubes, a very convenient structure for performing on-the-fly analysis of large volumes of data that has already proved its strengths in data warehouses [72]. Finally, a music data model, its algebra and a query language are presented in [90]. The data model is able to structure both the musical content and the metadata but does not address performance optimization issues. In particular, it does not provide an adequate framework to perform similarity based search. In [49], Jensen et al. address this issue and offer a multidimensional model that supports dimension hierarchies. We extend that multidimensional model by integrating fuzzy sets and addressing additional usage scenarios. Furthermore, this implementation proves to be able to handle a much larger music collection of a realistic size in the context of an MW.

The use of bitmaps in multidimensional databases is frequent. Different compression schemes exist to reduce the storage consumption of bitmaps. The Word Align Hybrid [92], WAH, and the Byte aligned Bitmap Compression [7, 8], BBC, are two very common compression algorithms. BBC offers a very good compression ratio and performs bitwise logical operations efficiently. WAH performs bitwise operations much faster than BBC but consumes more storage space. We propose a modified version of WAH compression technique to represent fuzzy sets. We show how fuzzy set operators can be adapted to directly manipulate the compressed representations in order to preserve the performance.

Significant efforts have been made in representing imprecise information in database models [26]. Relational models and object oriented database models have already been extended to handle imprecision utilizing the fuzzy set theory [75, 15]. This chapter proposes pragmatic solutions to store and manipulate fuzzy sets within multidimensional data cubes. While our focus is on musical data, we believe our approach can easily be generalized to the similarity matrices extensively used in fuzzy databases, e.g., to perform fuzzy joins.

4.3 Usage Scenario

The data obtained from a music recommendations system has to be organized to answer specific queries. Examples of such query scenarios are presented below.

4.3.1 The User Feedback

The user's opinion about the system's previous recommendations is a valuable piece of information for improving the future suggestion, e.g., by reinforcement learning. For each song played, the user can grade if the suggestion was wise based on the criteria provided, referred to as the query context. The query context can be the artist similarity, the genre similarity, the beat similarity, or any other similarity measure available to the system to perform a selection. The grading reflects if a proposed song was relevant in the given query context. For example, it is possible to retrieve the list of songs Mary liked when she asked for a list of rock songs or the ten songs she liked the most when she asked for similar songs to a song made by "U2".

Typically, the data obtained should contain:

1. a reference to the profile of a registered user in the system;
2. a reference to a query context provided by the user; and
3. the list of songs and marks so that for each song proposed, the user can grade how much she liked a particular song being part of the proposition.

Grades are given on a per song basis, they reflect if the user believes the song deserves its place among the suggested list of songs: strongly disagrees, neutral, likes, and loves. While the grade must not be a numerical value, we assume that a mapping function to the interval $[0, 1]$ exists so that when a user believes a song definitely deserves its place in the list, a high value in the interval should be given.

4.3.2 The User Preferences

Regardless of any given query context, some songs should never be proposed to Mary as she simply can't stand them or, on the contrary, some songs should be proposed more often as they are marked as Mary's favorites. Therefore, recommendation systems often offer to their users the possibility to rate any song on a *fan-scale* ranging from "I love it" to "I hate it" depending if they like the song or not. Such information is useful for building network based on users having similar musical taste. The database backend of the recommendation system should be able to find users similar to Mary based on his favorite and loathed songs.

The User Musical Preferences contains two different pieces of information:

1. a reference to a user registered; and

2. a list of songs associated with their respective grades on the fan-scale.

As above, we assume the mapping to the interval $[0, 1]$ so that if Mary hates a song, a low score is assigned; and if she loves it, a value close to 1 should be used. So, musical profiles can be used to modify the frequency a given song appears as a recommendation and build recommendation based on profile similarities.

4.3.3 The Songs Similarities

Finally, music recommendation system should be able to compare songs. For each pair of songs, the system is able to provide a similarity value with respect to a given aspect of the song such as the release year, the genre, the theme, the lyrics, or the tempo. The similarity values should indicate if two songs are "very different", "different", "somewhat similar", or "very similar" from the perspective of any given aspect of the song. For example, the song We will rock you by Queen is "very different" from the song Twinkle, twinkle little star with respect to their *genre similarity aspect.*

To compare songs, three pieces of information are necessary:

1. a pair of compared songs;

2. a similarity function that maps to a pair of songs to a similarity value; and

3. a similarity value reflecting how similar the two songs are.

Again, we assume that the similarity values can be mapped to the interval $[0, 1]$ so that, if two songs are very different, a value close to 0 should be used, and if they are very similar, a value close to 1 should be used instead. The scenario is very generic; very few assumptions are made about the properties of the functions used to compute the similarity values. In particular, the similarity functions do not have to fulfill the mathematical properties of a metric: the non-negativity, the identity of indiscernibles, the triangular inequality, and the symmetry properties. They do not have to be defined over the whole domain of song pairs. This allows similarities to be based on a wide diversity of song attributes.

4.4 An Algebra for Fuzzy Song sets

In this section, we introduce song sets as well as operators and functions to manipulate them.

Let X be the set of all songs. Then, a fuzzy song set, A, is a fuzzy set defined over X such that

$$A = \{\mu_A(x)/x : x \in X, \mu_A(x) \in [0,1]\} \tag{4.1}$$

and is defined as a set of pairs $\mu_A(x)/x$, where x is a song, $\mu_A(x)$, referred to as the membership degree of x, is a real number belonging to $[0,1]$, and $/$ denotes the association of the two values as commonly expressed in the fuzzy logic literature [41]. When $\mu_A(x) = 0$, song x does not belong to A, and when $\mu_A(x) = 1$, x completely belongs to A.

4.4.1 Operators

The following operators are classically used in order to manipulate song sets. They form a closed algebra.

4.4.1.1 Equality

Let A and B be two fuzzy song sets. A is equal to B iff for all song the membership degree of a song in A is equal to the membership degree of the same song in B.

$$A = B \Leftrightarrow \forall x \in X, \mu_A(x) = \mu_B(x) \tag{4.2}$$

4.4.1.2 Subset

Let A and B be two fuzzy song sets. A is included in B iff for all song, the membership degree a song in A is lower than the membership degree of the same song in B.

$$A \subseteq B \Leftrightarrow \forall x \in X, \mu_A(x) \leq \mu_B(x) \tag{4.3}$$

Note that the empty fuzzy song set defined with the null membership function, i.e., $\forall x \in X, \mu(x) = 0$, is a subset of all fuzzy sets.

4.4.1.3 Union

Let A and B be two fuzzy song sets over X. The union of A and B is a fuzzy song set with, for each song, a membership degree equal to the maximum membership degree associated to that song in A and B.

$$\begin{aligned} A \cup B &= \{\mu_{(A \cup B)}(x)/x\} \\ \mu_{(A \cup B)}(x) &= \max(\mu_A(x), \mu_B(x)) \end{aligned} \tag{4.4}$$

4.4.1.4 Intersection

Let A and B be two fuzzy sets over X. The intersection of A and B is a fuzzy song set with, for each song, a membership degree equal to the minimum membership degree associated to that song in A and B.

$$\begin{aligned} A \cap B &= \{\mu_{(A\cap B)}(x)/x\} \\ \mu_{(A\cap B)}(x) &= \min(\mu_A(x), \mu_B(x)) \end{aligned} \tag{4.5}$$

4.4.1.5 Negation

Let A be a fuzzy sets over X. The negation of A is a fuzzy song set with the membership degree of each song equal to its symmetric value on the interval $[0, 1]$.

$$\neg A = \{1 - \mu_A(x)/x\} \tag{4.6}$$

The following new operators are introduced specifically to manipulate song sets.

4.4.1.6 Reduction

Let A be a fuzzy set over X. The reduction of A is a subset of A such that membership degrees smaller than α are set to 0.

$$\begin{aligned} \text{Reduce}_\alpha(A) &= \{\mu_{A_{\overline{\alpha}}}(x)/x\} \\ \mu_{A_{\overline{\alpha}}}(x) &= \begin{cases} \mu_A(x) & \text{if } \mu_A(x) \geq \alpha, \\ 0 & \text{if } \mu_A(x) < \alpha \end{cases} \end{aligned} \tag{4.7}$$

The reduction operator changes the membership degree of songs below a given threshold to 0. It allows the construction of more complex operators that allow the reducing the membership degree granularity over ranges of membership degrees.

4.4.1.7 Top$_k$

Let A be a fuzzy set over X. The Top_k subset of A is a fuzzy song with the membership degree of all elements not having the k highest membership degree set to 0 and the membership degree of the k highest elements of A set to their respective membership degree in A.

$$\begin{aligned} \text{Top}_k(A) &= \{\mu_{\widehat{A^k}}(x_i)/x_i| \\ &\forall x_i, x_j \in X, 1 \leq i < j, \mu_A(x_i) \geq \mu_A(x_j)\} \\ \mu_{\widehat{A^k}}(x_i) &= \begin{cases} \mu_A(x_i) & \text{if } i \leq k, \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{4.8}$$

Note that the Top_k subset of A is not unique, e.g., when all elements have an identical membership degree. The Top_k operator returns a fuzzy song set with all membership degrees set to zero except for k elements with the highest membership degrees that remain unchanged. Top_k is a cornerstone for the development of complex operators based on relative ordering of the membership degrees.

4.4.1.8 Average

Let $A_1, \ldots, A_i$ be i fuzzy song sets. The average of $A_1, \ldots, A_i$ is a fuzzy song set that assigns to each song a membership degree equal to the arithmetic mean of the membership degrees of that song in the given sets.

$$\begin{aligned} \text{Avg}(A_1, \ldots, A_i) &= \{\mu_{avg(A_1,\ldots,A_i)}(x)/x\} \\ \mu_{Avg(A_1,\ldots,A_i)}(x) &= \frac{\sum_{j=1}^{i} \mu_{A_j}(x)}{i} \end{aligned} \tag{4.9}$$

The average operator in fuzzy sets is the pendant of the common average operator and is very useful to aggregate data, a very common operation in data warehousing in order to gain some overview over large datasets.

4.4.2 Defuzification Functions

The following functions are defined on song sets. They extract information from the song sets to real values or crisp sets.

4.4.2.1 Support

The support of A is the crisp subset of X that includes all the elements having a non-zero membership degree in A.

$$\text{Support}(A) = \{x \in X : \mu_A(x) > 0\} \tag{4.10}$$

4.4.2.2 Cardinality

The cardinality of A is the sum of the membership degrees of all its elements.

$$\#A = \sum_{x \in X} \mu_A(x) \tag{4.11}$$

4.4.2.3 Size

The size of A is the number of elements in A with non-zero membership degree, i.e., the size of the support set of A.

$$|A| = |\{x \in X : \mu_A(x) > 0\}| \tag{4.12}$$

4.4.2.4 Distance

The Minkowski distance of order $p \in \overline{\Re}$ between two song sets is defined as follows.

$$d_{p\geq 1}(A, B) = \sqrt[p]{\sum_{x \in X} |\mu_A(x) - \mu_B(x)|^p} \tag{4.13}$$

The 1-norm distance is the Manhattan distance, the 2 -norm distance is the Euclidean distance, and the ∞ -norm is the Chebyshev distance.

4.5 The Music Warehouse Cubes

In this section, we present two data cubes to store the information presented in the scenarios. For each cube, the fuzzy song sets are used conformingly to Section 4.4.

4.5.1 The Song Similarity Cube

The Song Similarity cube captures similarity between songs with respect to selected similarity functions. The cube is composed of two dimensions: a song dimension and similarity dimension; they are represented in Figure 4.1. The song dimension captures all the details about a song, including editorial information such as the artist name, the publication year or any acoustic information such as the beat of the song or its genre. For each of these attributes, similarity functions can be created, e.g., an artist similarity function that gathers information from external web sites and social networks, or a similarity function that compares the genre wherein songs have been classified, aware that some genres are more similar than others, or the timbre comparison that uses low-level extracted information to provide a full comparison matrix.

Each dimension has a hierarchy, which defines how the data can be aggregated to provide different degrees of granularity, e.g., the similarity of songs between subgenres and the similarity of songs between coarsely defined genres. Similarity function of coarser granularity can also span over different attributes, e.g., to provide some average similarity values out of attributes obtained using different extraction algorithms.

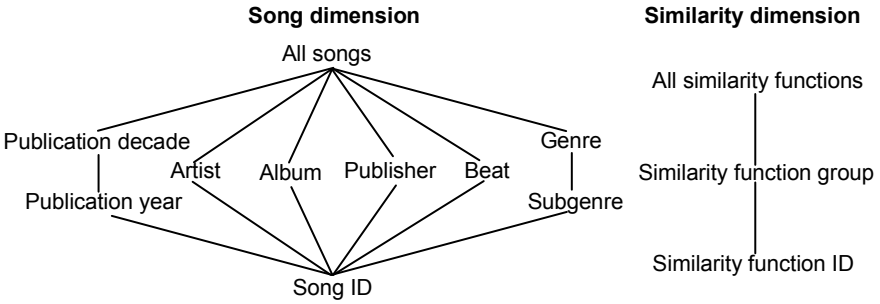


Figure 4.1: Closest Songs Cube Dimensions

At its most detailed level, the cube is organized based on a star schema, using three tables: the song dimension table, the similarity function table and the closest songs fact table. The closest songs fact tables is composed of three attributes: a reference to a song (referred to as the seed song), a reference to a similarity function, and a fuzzy song set. The notion of similarity between a song and the seed song is represented by the fuzzy song set membership degree. The closest songs take a high membership degree while the farthest songs have a low membership degree. Some usage examples of the Song Similarity Cube populated with the data in Table 4.1 are presented below.

Typical queries involve the intersection, union, and reduction operators. The queries can be performed on the song seeds using pieces of information such as the artist or the creation year. Closest Songs Cube usage examples are presented below. The example assumes the creation of a new SQL data type, called FZSET, using object-relational extensibility functionality like found in PostgreSQL [86]. For example, the closest songs attribute in the fact table is of type FZSET. The FZSET implementation details will be discussed further.

Example 1: "What are the songs that have a similar beat to the song "One" by U2?"

```
SELECT SUPPORT(REDUCE(0.6, c.songs)
FROM closest_songs c
INNER JOIN songs as a USING (song_id)
INNER JOIN similarity_functions as b USING (c.sim_id)
WHERE a.title = "one" AND a.artist = "U2" and b.sim = "beat 1"
```

In a star schema, the fact table and the 2 dimensions tables are joined to form the cube. As shown in Example 1, retrieving the similarities between a song and all the others simply requires selecting a song and a similarity function from the dimension

Song dimension

song id	title	artist	Jazz	Rock	Beat
1	We will rock you	Queen	Low	High	Medium
2	One	U2	Low	Medium	Medium
3	Hips Don't Lie	Shakira	Low	Low	High

Similarity function dimension

sim id	sim func	sim group
1	Rock	Acoustic
2	Jazz	Acoustic
3	Beat	Acoustic
4	Artist	Editorial

Closest songs fact

song id	sim id	closest songs
1	1	{ 1/1; 0.5/2; 0/3 }
2	1	{ 1/2; 0.3/1; 0.2/3 }
3	1	{ 1/3; 0.6/2; 0.4/1 }
1	2	{ 1/1; 0.2/2; 0.1/3 }
2	2	{ 1/2; 0.9/1; 0.2/3 }
3	2	{ 1/3; 0.8/2; 0.7/1 }

Table 4.1: Data of the Song Similarity Cube

tables and retrieving the corresponding FZSET in the closest song table. The support function transforms an FZSET data type into a regular SQL crisp set of elements having non-zero membership degrees.

In Example 2, the mu function returns the membership value associated to a given element. The similarity between two songs can be obtained by retrieving the full fuzzy song set representing song similarities for the first song, and filtering out the results to only return the element matching the second song. However, with such an operation being so common, optimization based on the physical storage structure of the fuzzy song set can be performed, thus motivating the need for creating a specific element search function within a fuzzy song set.

Aggregation functions allow multiple fuzzy song sets to be retrieved and combined. In Example 3, multiple songs are matching the selection criteria in the song dimension, causing multiple fuzzy song sets to be retrieved from the closest song table. The fuzzy song sets are then combined using the union operator; finally the elements with the 100 highest membership degrees are returned.

User dimension

user	country	age	favorite songs
John	USA	52	{ 0.8/1; 0.6/2; 0.3/3 }
Alice	Spain	41	{ 0.9/2; 0.5/1; 0.3/3 }
Maria	Greece	28	{ 0.6/1; 0.3/2; 0.1/3 }
Bob	Denmark	22	{ 0.1/1; 0.7/2; 0.7/3 }

Query dimension

query id	query
1	return some rock music
2	return some traditional music
3	return some latin american music

User Feedback fact

user	query id	feedback
John	1	{ 1/1; 0.5/2; 0/3 }
John	2	{ 1/2; 0.3/1; 0.2/3 }
Alice	1	{ 1/3; 0.6/2; 0.4/1 }
Alice	3	{ 1/1; 0.2/2; 0.1/3 }
Maria	1	{ 1/2; 0.9/1; 0.2/3 }
Bob	2	{ 1/3; 0.8/2; 0.7/1 }

Table 4.2: Feedback Cube

Example 2: "Find the beat similarity between two songs; the first song is identified with the artist, album, and title attributes from the song dimension, the second is identified using its unique key."

```
SELECT MU(c.songs,el)
FROM closest_songs c
INNER JOIN songs as a USING (song_id)
INNER JOIN similarity_functions as b USING (sim_id)
WHERE a.artist = "U2"
AND a.album= "Achtung Baby"
AND a.title= "One"
AND b.sim = "beat 1"
GROUP BY a.album_id
```

As in a spreadsheet, aggregation can be performed on both dimensions. Example 4 retrieves all the versions of a song in the different albums of an artist and returns

Example 3: "Retrieve the 100 songs having the most similar beat to the songs made by U2."

```
SELECT SUPPORT(TOP(100, UNION(c.songs))
FROM closest_songs c
INNER JOIN songs as a USING (song_id)
INNER JOIN similarity_functions as b USING (sim_id)
WHERE a.artist = "U2" AND b.sim = "beat 1"
GROUP BY a.album_id
```

Example 4: "Return the similar songs to the given song across the different beat similarity functions available."

```
SELECT SUPPORT(AVG(songs))
FROM closest_songs c
INNER JOIN songs as a USING (song_id)
INNER JOIN similarity_functions as b USING (sim_id)
WHERE a.title = "one" AND a.artist = "U2" and b.sim = "beat"
GROUP BY a.albumid, b.similarity_function_group
```

an average over similarity functions of the same type, such as the beat, the genre, or the mood.

4.5.2 The User Feedback Cube

The User Feedback Cube collects relevance statistics about the songs proposed to users by the music recommendation system. As illustrated by Figure 4.2, the User Feedback Cube is composed of two dimensions: the user dimension and the query dimension. For each user and query, the user feedback is stored. The feedback given for a particular played song is stored as a membership degree representing how relevant the proposed song is in the context of the query. A very low membership degree is given when a user believes the song should not have been proposed. The Feedback and the Favorite Songs attributes are both defined using the FZSET abstract data type. The user dimension is composed of a hierarchy allowing users to be aggregated along the various attributes composing their profiles. One of these attributes is a fuzzy song set representing the users favorite songs; it becomes thus simple to compare groups of users created based on the users musical tastes. The hierarchy on the query dimension permits to obtain overview along group of semantically close queries.

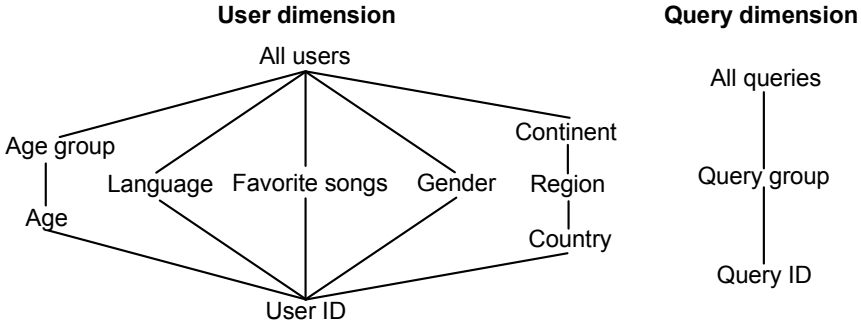


Figure 4.2: Dimensions Composing the User Feedback Cube

Example 5: "What are the favorite songs three users have in common?"

```
SELECT SUPPORT(REDUCE(0.8, INTER(Favorite songs))
FROM users
WHERE user_id = "1" OR user_id = "2" OR user_id = "3";
```

As shown in Example 5, retrieving the songs three users like is an immediate query using the proposed algebra; only the user dimension table is required. Here, the aggregation form of the intersection function allows straight-forward selection of the intersection between three multiple sets. The Reduce operator selects only the songs resulting from the intersection with a membership degree above 0.8. The support operator transform the fuzzy song set object into a crisp set that can be manipulated with the regular SQL algebra.

Example 6: "Who are the 100 users that have the most similar taste to Johns taste?"

```
SELECT b.user_id
FROM users as a, users as b
WHERE a.user_id = "1"
ORDER BY distance(a.favorite_songs, b.favorite_songs) ASC
LIMIT 100;
```

Example 6 illustrates how, using a self-join, the user dimension can be used to findsimilarities between users based on their favorite songs.

In Example 7, using the user dimension, only the users born in the 80s are selected, and the average feedback per query type is then calculated. Again, using the reduce and support operators, only the songs with a high membership degree are output as crisp sets.

Example 7: "Per query type, what are the songs users born in the 80s were usually happy to hear?"

```
SELECT SUPPORT(REDUCE(0.8, AVERAGE(uf.feedback)), q.query_type
FROM user_feedbacks as uf
INNER JOIN users as u USING (user_id)
INNER JOIN queries as q USING (query_id)
WHERE "1 JAN 80" <= u.DOB AND u.DOB <= "31 DEC 89"
GROUP BY q.query_type;
```

Example 8: "What are the 100 songs that fans of Elvis liked the most when they asked for Rock songs?"

```
SELECT SUPPORT(TOP 100(AVERAGE(uf.feedback)))
FROM user_feedbacks as uf
INNER JOIN queries as q USING (query_id)
WHERE u.user_id IN (
SELECT user
FROM songs
WHERE SUPPORT(TOP(10,favorite song)) = song_id
AND artist = "Elvis"
) AND q.query = "Rock songs"
```

Example 8 performs an aggregation of the user feedback. The selection of the users for the aggregation is performed using the favorite songs in the user dimension. Thus, both fuzzy song sets in the user dimension table and the fact table are used.

4.6 Storage

In this section, three different storage options for representing fuzzy song sets in the MW are presented: tables, arrays, and bitmaps. A prototypical MW where song elements are uniquely identified using 32 bits is used to illustrate the discussion. The proposed MW can reach a size of over 4 billion songs and at least 100 different membership degrees.

4.6.1 Table

The first solution is to represent the fuzzy song set attribute as a table with three columns: *(seed song, song, membership degree)*. Let s be the size of the seed song set, e the size of the song set, and m the size of the set of all the values the membership

degree can take. The size of the payload, i.e., the size of the data when not considering the overhead due to the DBMS, denoted p, can be calculated as follows.

$$p = s \cdot e(\log_2 s + \log_2 e + \log_2 m) \tag{4.14}$$

where $\log_2 s$, $\log_2 e$, $\log_2 m$ are the minimum number of bits required to store respectively a seed song, a song, and a membership degree.

The quadratic growth can be limited by admitting only k songs for each seed song to be physically stored in the table and letting the remaining songs take a default membership degree. The selection of which song should be represented is dependent on the application. Here, we assume that the elements with the highest membership degree are interesting; this is performed using the Top_k operator. The size of the payload can then be estimated as follows.

$$p = s \cdot k \cdot (\log_2 s + f \cdot (\log_2 e + \log_2 m)) \tag{4.15}$$

When 2^{32} seed songs are present, the database reaches its maximum capacity. In such case, the size of the payload, if only the 1000 elements with the highest membership degree are physically stored, reaches 36 TB. On a data set composed of 10,000,000 seeds, the payload attains 84 GB.

4.6.2 Array

A second approach is to use one-dimensional arrays containing the songs and their associated membership degrees for representing fuzzy song sets. The data is stored in a table with two columns: *(seed song, array)*. As with tables, only the $k \leq e$ most similar songs should be physically stored. The size of the payload grows as follows.

$$p = s \cdot (\log_2 s + k \cdot (\log_2 e + \log_2 m)) \tag{4.16}$$

When storing the 1000 closest songs of 2^{32} song seeds, the size of the payload is reduced to 19 TB; on a data set composed of 10,000,000 song seeds, the payload reaches a size of 44 GB However, since the probability of having no songs for a particular membership degree is small, ordering the fuzzy song set by membership degrees allows membership degrees to be stored using one bit relatively to each other: a bit set means to move to the next lower membership degree, a bit unset means to keep the same membership degree. In the unlikely case of a gap in the sequence of membership degrees, a dummy element, referred to as the empty element, is used to jump to the next membership degree. For large gaps, successive empty elements are used as shown in Figure 4.3.

For example, the fuzzy song set $\{100/1234, 100/2345, 99/3456, 97/4567, 96/5678\}$ is represented by the array $[\{1234, 100\}, \{2345, 100\}, \{3456, 99\}, \{4567, 97\}, \{5678, 96\}]$

seed	elem	elem	elem	elem	elem	elem	elem	elem
	elem	elem	gap	elem	elem	elem	elem	elem
	elem	elem	elem	elem	gap	elem	elem	elem
	elem	gap	elem	elem	elem	elem	elem	elem

Figure 4.3: Closest Songs Cube Dimensions

that is compressed as $[\{1234, 0\}, \{2345, 1\}, \{3456, 1\}, \{0, 1\}, \{4567, 1\}, \{5678, 0\}]$, where only one bit is required to capture a decrement of the membership degree, and 0 is the empty element.

The compression ratio, r, obtained is as follows.

$$\mathrm{r} = \frac{k \cdot (\log_2 m + \log_2 e)}{(k + x)(\log_2 e + 1)} \tag{4.17}$$

In order to be efficient, i.e., $r > 1$, the number of empty elements in the data set has to remain limited.

$$x < k \cdot \frac{\log_2 m - 1}{\log_2 e + 1} \tag{4.18}$$

The compression ratio in the best (no empty element) and worst ($m - 1$ empty elements) case scenarios are:

$$\begin{aligned} r^- &= k \cdot \frac{(\log_2 m + \log_2 e)}{(k + m - 1) \cdot (\log_2 e + 1)} \\ r^+ &= \frac{\log_2 m + \log_2 e}{\log_2 e + 1} \end{aligned} \tag{4.19}$$

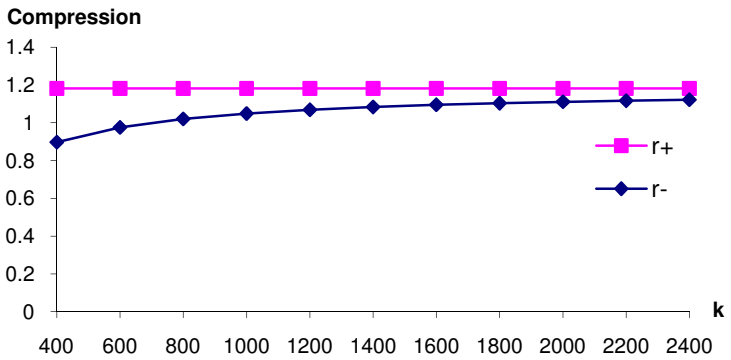


Figure 4.4: Best and Worst Compression Ratio for the Arrays

For high k values, the likelihood of using empty elements vanishes, therefore causing r^- to asymptotically converge to r^+ as k increases. Figure 4.4 shows the compression ratio r^+ and r^- for membership degrees represented on 7 bits (128 different values), and fuzzy song set and song seeds represented using 32 bits. For $k = 1000$, the compression ratio ranges between 1.04 and 1.18. The full similarity matrix represented with compressed arrays takes 17 TB.

4.6.3 Bitmap

A third option is to use bitmaps to represent fuzzy song sets. In a bitmap [21], each element is represented by a position in a sequence of bits. Typically, in a bitmap index, a bitmap for each attribute value is created. The size of each bitmap is equal to the cardinality of the indexed elements. Fuzzy song sets can be constructed using the same structure. A fuzzy song set is composed of a bitmap for each membership degree an element can have. As illustrated in Figure 4.5, each song element is represented with a bit set in the bitmap corresponding to its membership degree.

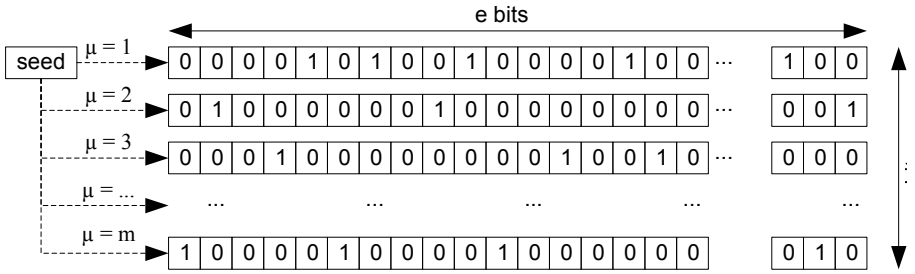


Figure 4.5: Representation of a Fuzzy Song Set with an Array of Bitmaps

The size of the payload can be estimated as follows.

$$p = s \cdot (\log_2 s + f \cdot e) \tag{4.20}$$

The bitmap size can be dramatically reduced using compression algorithms. The Word-Aligned-Hybrid (WAH) bitmap compression method offers a good compression ratio on sparse bitmaps while preserving query performance [92].

Briefly, in a WAH-compressed bitmap, the bitmap is divided in 32 bit long words. The first bit of each word is used to mark if the word is a literal word or a fill word. If the first bit of a word starts with a unset bit, the word is a literal word; the remaining bits are then used to store a classical 31 bit long bitmap. A fill word starts with a set bit and indicates the presence of a run composed of homogeneous 31 bit long groups of set or unset bits; thus, fill words are of two kinds: 0-Fills or 1-Fills. The second

bit of a fill word is used to differentiate runs of unset bits from runs of set bits. The remaining 30 bits are used to count the number of homogeneous 31 bit long groups the run contains.

Figure 6 shows an example of how the bitmap composed of 9×0, 3×1, 56×0, 69×1, 98×0, 3×1, and 6×0 can be compressed using WAH. First, the uncompressed bitmap is divided into groups of 31 bits. If a group forms a literal word, an unset bit is prepended to it. Otherwise, the group is replaced by an appropriate fill word and a counter of the number of identical consecutive groups following the current group.

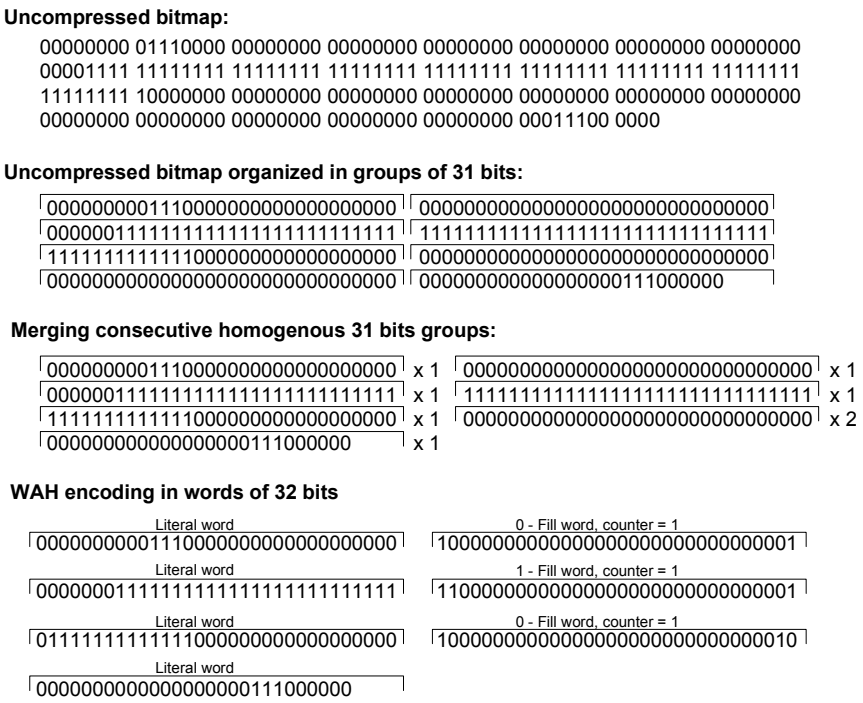


Figure 4.6: The WAH Bitmap Compression

The WAH compression becomes very efficient when many consecutive zeros or ones can be represented with fill words. In the worst bit distribution, i.e., a random bitmap, the WAH algorithm reduces the size of the bitmap as follows.

$$p_{\text{WAH}}(n, d, w) \approx \frac{w \cdot n}{w - 1} \left(1 - (1 - d)^{2w-2} - d^{2w-2}\right) \text{ b} \tag{4.21}$$

where n is the size of the bitmap in bits, d is the bit density, i.e., the fraction of bits set, and w is the word length, 32 on most computers. Using the Top_k operator, the

bit density is $d = k/e$. On a fuzzy song set of 2^{32} songs where only 1000 closest songs are physically stored, $n = 2^{32}$, and $d = \frac{1000}{2^{32}}$. The size of each bitmap reaches 63883 b.

As previously illustrated by Figure 4.6, a bitmap is constructed for each of the membership degree a song element can possibly take. The fuzzy song set is then represented using an array composed of 100 bitmaps, but this does not affect the size of the overall bitmap as the bit density of in each bitmap will proportionally decrease, maintaining the bit density in the full bitmap unchanged.

$$p \approx s \cdot \left(\log_2 e + f \cdot p_{\text{WAH}}(e \cdot m, \frac{k}{e \cdot m}, w) \right) \tag{4.22}$$

In an MW of 2^{32}songs, where 1,000 song elements with the highest membership degree are physically stored, the size of the payload reaches 33 TB. On a data set composed of 10,000,000 song seeds, the payload size is 76 GB.

The size of the compressed bitmap for each song seed is only slightly increased to 63999 b. Therefore, in an MW of 2^{24} song seeds with one fuzzy song set attribute, the size of the database reaches 140 GB.

4.6.4 Payload Estimates Comparison

Figure 4.7 shows the expected size for storing a Fuzzy Song Set Attribute (FSSA) for each of the 2^{32} song seeds and for different values of k. The linear growth of the WAH bitmap with the number of stored elements is explained by considering $k/n << 1$ and applying a binomial decomposition. The payload can then be approximated by $p_{\text{WAH}} \approx 2 \cdot k \cdot w$.

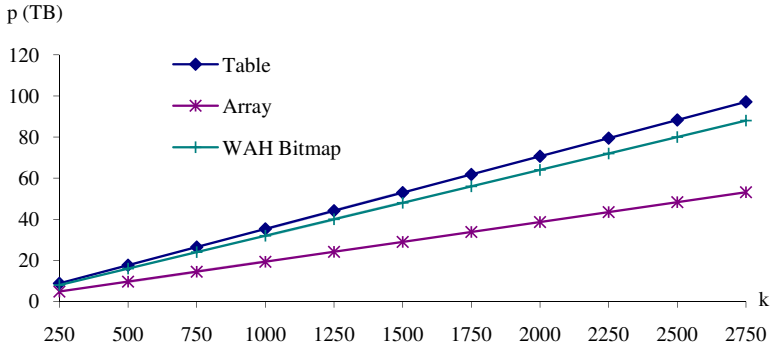


Figure 4.7: Estimate of the Payload Storage Requirements

In arrays, the seed elements only have to be stored once per FZSET. Arrays take thus half the storage requirements of tables. With arrays, however, the data need to

be compressed and reorganized, thus leading to an overall increase in complexity. The array compression scheme is focused on compressing the membership degree. The compression occurs on the 7 bits used to represent the membership degree but leave the 32 bits representing each element untouched; thus limiting the maximum compression performance that can be achieved. Bitmaps, on the other hand, are focused in compressing the 32 bits representing the elements; this is done by imposing a position to each song element. These important structural differences will have an impact on the implementation of operators and functions.

4.6.5 Storage Estimates and Benchmark

This section describes the storage requirements for the implementation of the Song Similarity Cube fact table. Therefore, some parts of the following are dependent on the DBMS chosen for implementing the cube. We calculate some storage requirements estimates for each of data structure. As our estimates match experimental results, we proceed on predicting the size of each storage option depending on the number of fuzzy elements they contain. The experience was conducted on PostgreSQL 8.3, well-known for its scalability.

As previously explained, the songs can be uniquely identified using 32 bits and the membership degree of each song element has a granularity of 100. The dataset used for the implementation consists of 150,834 songs, gathered from the Intelligent Sound project. Song similarities are computed using a genre classifier collecting acoustic features from a popular media player [56].

The expected table overhead in PostgreSQL can be estimated by considering tuple overhead and page overhead [86]. In our configuration, pages have a fixed size of 8 KB. Since tuples are not allowed to span over multiple pages, PostgreSQL uses secondary storage tables, referred to as The Oversized-Attribute Storage Technique (TOAST) tables, to store large attributes. Using TOAST, large field values are compressed and/or broken up into multiple physical rows. TOAST tables use the Lempel-Ziv, briefly LZ, compression technique to reduce their size [99]. The compression of toasted attributes being optional, we will compare the different possible setups.

In a table, the number of rows is the product of the number of seeds and the number of elements per seed: $150,834 \cdot 1000 = 150,834,000$ rows. Each page has a size of 8KB, with a header of 24 bytes, thus leaving 8,168 bytes of free space. Each row has a payload of $4 + 4 + 4 + 1 = 17$ bytes. Each tuple is stored after a 20 bytes long header, and is aligned to start on the 32nd byte. Therefore, the size of each row in the table is $31 + 17$ bytes. Thus, each page can accommodate 185 rows, and 150,834,000 rows will require 815,319 pages, thus taking a disk space of $815,319 \cdot 8$ KB $= 6,369.67$ MB. In our storage experiment on the 150,834 songs, gathered from the iSound database, this is exactly the storage size taken on disk; thus indicating that our estimate is precise.

For arrays, each element has to be aligned on 4 bytes, thus 8 bytes are necessary to store the element and the membership degree. Additionally, 4 bytes are used to store the size of the array. Each array has therefore a size of $4 + 4 + 1000 \cdot 8$=8008 bytes not allowing two tuples to fit on a single page. Therefore 150,834 pages of 8 KB are needed, causing the storage requirements to be 1,178 MB.

For bitmaps, in the worst case compression scenario, each of the 1,000 elements requires both a fill-word and a literal word, e.g., when a 0-fill word is required between each set bit. A word takes 4 bytes, thus 8 bytes per elements and 8,000 bytes per bitmap. For each bitmap, an additional 4 bytes long integer is required to keep track of the size of the data, thus adding $100 \cdot 4$ bytes. Thus a bitmap cannot fit on a page and has to be moved to an auxiliary toast table, where each bitmap is split into chunks of 2,000 bytes. In that case, 4 rows per bitmap attribute are required in the auxiliary table. Storage estimates show that in the most pessimist case 1,472 MB are required to store the bitmaps. In the selected dataset, 183,184 pages are required to store the bitmaps. The total space taken by the WAH compressed bitmap storage representation is therefore: 1,431 MB.

If the number of element increases, a similar storage technique using an auxiliary TOAST table is required for the array data structure. As with bitmaps, data larger than 2,000 bytes is split into 2,000 bytes chunks. Each array is therefore divided into 5 chunks, and $150,834 \cdot 5$ chunks are needed. For each data chunk, a 31 bytes long header has to be added. Since 8,168 bytes of storage are available per page, only 4 chunks can be stored per page and 188,543 pages are needed. The total size of the array data structure is 1,472 MB when stored using a TOAST table.

Further compression of TOAST data using standard LZ algorithm can be performed. The compression ratios are data depending. Table 4.3 shows the storage requirements for the three storage options. In addition, the space required to index seed songs and similarity functions using a standard B-Tree and storage requirements for LZ-compressed data are presented.

Our experiments show that the real size requirements match the estimates. While table are certainly the most straightforward solution, they are a bad choice for data storage requirements and indexing purposes. With respect to the payload, the arrays are very promising but suffer from an important overhead that makes arrays and WAH compressed bitmaps very comparable in term of storage size. Furthermore, since array elements are aligned on 8 bytes, compressing the array does not bring any storage benefit and adds unnecessary complexity. LZ compression works better on bitmaps, therefore creating a sensible difference in favor of bitmaps; this is observation might, however, be data dependent. Finally, with respect to the implementation of the two new data types, WAH-bitmaps are a more complicated data structure to build; the compression requires some particular attention and the variable length nature of the bitmap brings additional complexity.

		Size (MB)
Table	Payload est. (MB)	1,852
	Overhead est.	4,518
	Total est.	6,370
	Real size	6,370
	B-tree index size	3,231
	Total	**9,601**
Array	Payload est. (MB)	666
	Overhead est. (MB)	511
	Total est. (MB)	1,178
	Real size (MB)	1,178
	Real size + LZ (MB)	794
	B-tree index size(MB)	3
	Total	**1,181**
WAH Bitmap	Payload est. (MB)	1,151
	Overhead est. (MB)	296
	Total est. (MB)	1,447
	Real size (MB)	1,447
	Real size + LZ (MB)	719
	B-tree index size(MB)	3
	Total	**1,450**

Table 4.3: Storage Comparison

Using identical storage estimates, we predict the size of tables, arrays, and bitmap with respect to k. Considering that k elements are required in order for the data to be useful, we can thus choose what data structure is the most appropriate. The results of the size estimates are shown in Figure 4.8. For all values of k, tables are the worst solution. For $k > 2,000$, arrays and WAH-compressed bitmaps tend to behave very similarly. For lower values of k, due to the data organization in pages, results vary sensibly depending on k. However, arrays always keep a slight advantage.

4.7 Functions and Operators

In this section, we compare the array and bitmap storage structure with respect to the performances of their operators.

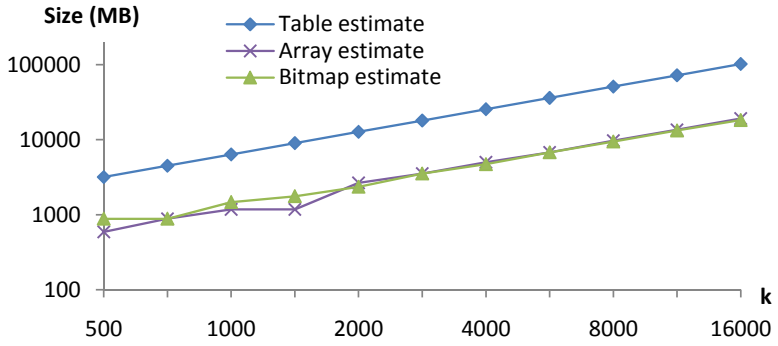


Figure 4.8: Estimate of the Payload Storage Requirements

4.7.1 WAH Bitmap Operations

The original WAH compression method has been slightly adapted in order to manipulate bitmaps of different lengths. First, the last word, i.e., the remainder of the uncompressed bitmap is stored as if the bitmap is extended with extra unset bits to finish the last word. So a bitmap composed of: $10 * 0$'s, $21 * 1$'s, and $4 * 1$'s becomes $< 001FFFFF > < 78000000 >$ and not $< 0001FFFF > < 0000000F >$ as in the original algorithm. This allows no particular treatment for the last word and allows expanding existing bitmaps without any extra manipulations. Logical operations on WAH-compressed bitmaps can be performed without decompressing the bitmaps. Operations are performed by scanning both inputs word by word. If two fill words are met, the result will be a fill word of type resulting from the operation; its length is the minimum length of the two input fill words. If two literal words, or a literal and a fill word are met, the result will be a literal corresponding to the operation.

4.7.2 Intersection and Union

The computation of the intersection or the union of two fuzzy song sets represented in arrays is performed by a modified sort-merge. The arrays are first decompressed and sorted by element. In our experiment, the sorting of the array with respect to its elements is done using the quicksort algorithm. Once sorted, the membership degrees of identical elements are compared. For an intersection, if both elements are present, the minimum membership degree is placed in the array; for a union, the maximum membership degree of both elements or the membership degree of the existing element are placed in the return array.

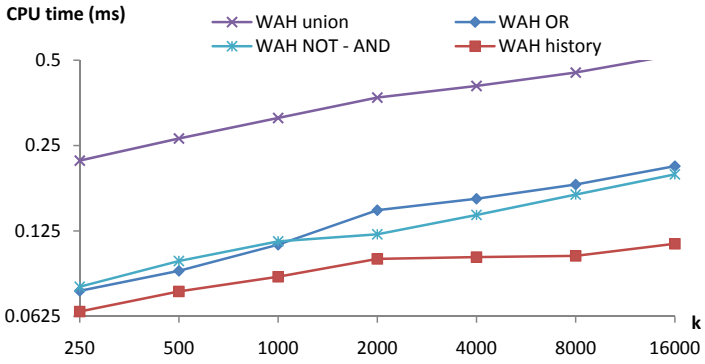


Figure 4.9: CPU Time Required for the Various Steps of a Union of Fuzzy Song Sets Represented with Bitmaps

The computation of the WAH union is performed as follows. In the WAH bitmap representation, the elements are organized per membership degree. For each membership degree starting from the highest, we perform a logical OR on the compressed bitmaps. To prevent future operations to set a bit already set previously for another membership degree, we have to maintain a history of bit, also represented using a WAH-bitmap. This costs two additional operations on the bitmaps, a compressed "NOT-AND" to check that a bit was not previously set, and a compressed or, to maintain the history up to date as we scan through the various membership degree.

Pseudo C code for performing the union is shown in Algorithms 4.1 and 4.2, . The computational cost of the "OR", the "NOT-AND", and the "OR" for maintaining the set bit history are shown in Figure 9. The WAH union is the sum of the three operations.

No update of the history is needed when handling the last bitmap, thus the CPU time reaches a ceiling when no more elements are added to a bitmap corresponding to a level higher than 1. After 2000 elements, all the bitmaps have elements. New elements are added in the last bitmap corresponding to the lowest membership degree.

For sparse bitmaps, the number of elements grows linearly with the number of elements. As the density of bits set increases, the proportion of literal words increases, thus increasing the likelihood of new element being added to existing literals rather than splitting fill words into literals. Figure 4.10 shows the average input and output length of the bitmaps used for benchmarking the CPU time of the "OR" operation. After 2000 elements, the length growth diminishes due to the increase in the number of literals.

The union of arrays is highly efficient for low numbers of elements. As expected, their performances decrease as the number of elements increases. Additionally, the

Algorithm 4.1 Bitwise logical AND operator on two WAH compressed bitmaps

structure wahrun:
- it ▷ iterator
- $data$ ▷ decompressed data
- $nWords$ ▷ group counter
- $isFill$ ▷ flag for fill words

run_decode(wahrun run, word)

1: **if** $word$ is counter **then**
2: $run.data \leftarrow (word >$ onecounter?allones : allzeros$)$
3: $run.nWords \leftarrow word\&mask\_counter$
4: $run.isFill \leftarrow 1$
5: **else**
6: $run.data \leftarrow word\&mast\_literal$
7: $run.nWords \leftarrow 1$
8: $run.isFill \leftarrow 0$

and(bitmap x, bitmap y, bitmap rtnBitmap)

1: nWords = 0 ▷ counter set to minimum
2: run xrun, yrun ▷ initialize to run structures
3: **while** $xrun.it < size(x)$ and $yrun.it < size(y)$ **do**
4: **if** $xrun.nWords = 0$ **then** ▷ load a new word from x
5: $run\_decode(xrun, x.last)$
6: **if** $yrun.nWords = 0$ **then** ▷ load a new word from y
7: $run\_decode(yrun, y.last)$
8: **if** $xrun.isFill$ and $yrun.isFill$ **then**
9: $nWords \leftarrow min(xrun.nWords, yrun.nWords)$
10: $appendFill(rtnBitmap, nWords, xrun.data\&yrun.data)$
11: $xrun.nWords \leftarrow xrun.nWords - nWords$
12: $yrun.nWords \leftarrow yrun.nWords - nWords$
13: **else**
14: $appendLit(rtnBitmap, xrun.data\&yrun.data)$
15: $xrun.nWords \leftarrow xrun.nWords - 1$
16: $yrun.nWords \leftarrow yrun.nWords - 1$
17: **if** $xrun.nWords = 0$ **then**
18: $xrun.it \leftarrow xrun.it + 1$
19: **if** $yrun.nWords = 0$ **then**
20: $yrun.it \leftarrow yrun.it + 1$

sort operation significantly increases the computation time. Note, however, that the resulting set is sorted, thus preventing successive sort operations to be necessary, e.g., in case the function is used for an aggregation. But, even in the best case scenario, when no sorting of the elements is required, the CPU time spent on the union of arrays is proportional to the number of elements in the sets. Bitmap operations, however, are linearly proportional to the number of words in the input bitmap and not directly to

Algorithm 4.2 Union of fuzzy song sets represented with two arrays of WAH compressed bitmaps with membership degree ranging from 0 to 100

$bitmap * union(bitmap * x, bitmap * y)$
$bitmaptmp, history$ ▷ temporary and history bitmaps
$bitmapz[100]$ ▷ z is the return array of bitmap
mu ▷ membership degree
1: **for** mu ranging from 100 to 2 **do**
2: $tmp \leftarrow or(x[mu], y[mu])$ ▷ logical or, saved in tmp
3: $z[mu] \leftarrow notand(history, tmp)$ ▷ check with history
4: $history \leftarrow or(history, tmp)$ ▷ update history
no history update for mu = 1
5: $tmp \leftarrow or(x[mu], y[mu])$ ▷ logical or
6: $z[1] \leftarrow notand(history, tmp)$ ▷ check with history
7: **return** z;

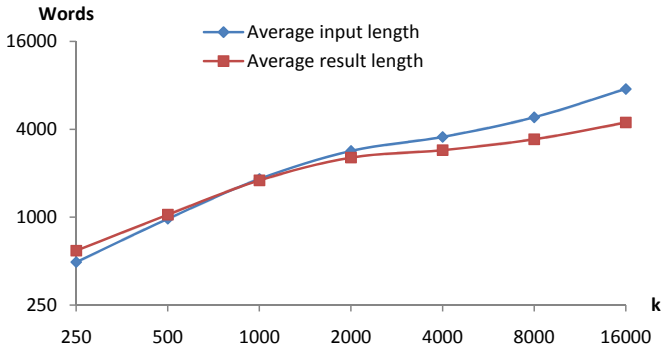


Figure 4.10: Average Bitmap Input and Output Length Depending on the Number of Song Elements Stored

number of elements, i.e., the number of bits set. As the number of elements increases, bitmaps will keep a nearly constant processing time where arrays will be proportional to the number of elements. Efficiency of the array and bitmaps union operations on the song similarity dataset is shown in Figure 4.11.

4.7.3 Top

The top operation for the array data structure requires ordering the elements with respect to their membership degrees. Since the number of membership degrees is limited, the sort is performed using a bucket sort whose complexity is linear in the number of elements.

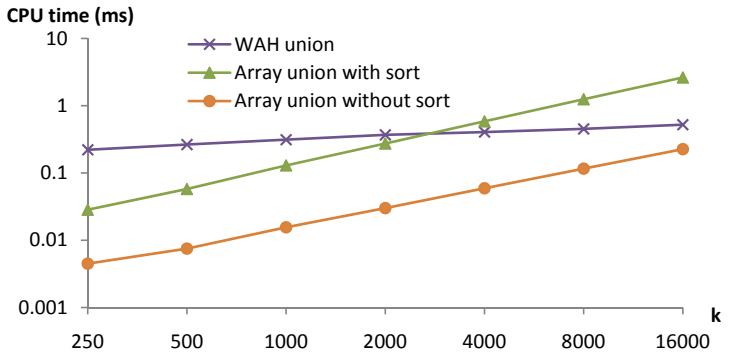


Figure 4.11: Comparison between the Performances of the Union Operator for Arrays and WAH Bitmaps

For WAH bitmaps, the elements are already grouped by membership degree. The only operation required is to scan the compressed bitmap, starting with the highest membership degree. As soon as k elements are found, the scan stops. The number of operations is thus only depending on the number of words needed to be read during the scan before k set bits are found. Unlike arrays, the operation is independent from the total number of elements in the bitmaps. Pseudo C code for performing the top operation is shown in Algorithm 4.3.

Finally, returning the resulting WAH bitmaps is performed simply by copying the input bitmaps and truncating it at the right place. Sorting the array is a slower process as it requires copying elements one by one. The CPU time spent for performing top operations depending on the size of the fuzzy song set are shown in Figure 4.12.

4.7.4 Reduce

On an array, the reduce operation requires scanning the elements of the array; the computational cost is therefore proportional to the number of elements. In a WAH bitmap, since the elements are already organized per membership degree, the operation only consists of deleting the bitmaps corresponding to membership degree lower than alpha from the input bitmap. Pseudo C code for performing the reduce operation is shown in Algorithm 4.4. The computation time results are shown in Figure 4.13.

4.8 Generalization to Other Domains

The generalization from fuzzy song sets to other domains with respect to the storage solutions is immediate for both arrays and WAH bitmaps.

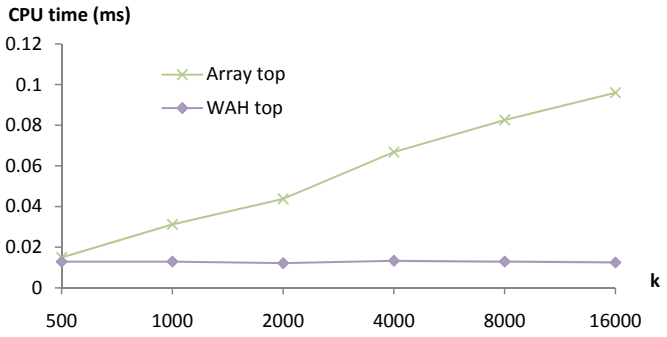


Figure 4.12: Comparison between the Performances of the Top Operator for Arrays and WAH Bitmaps

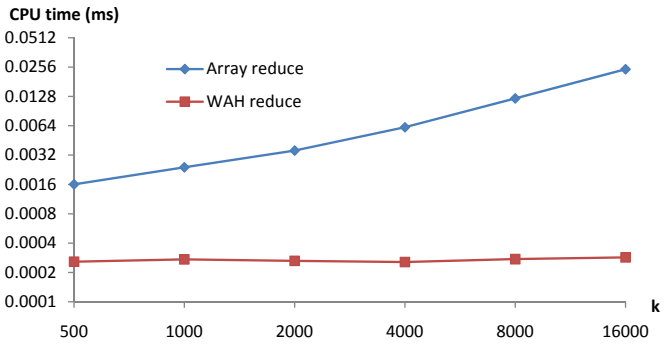


Figure 4.13: Comparison between the Performances of the Reduce Operator for Arrays and WAH Bitmaps

Algorithm 4.3 Top operation of a fuzzy song set represented by an array of WAH compressed bitmaps with membership degree ranging from 0 to 100

```
   bitmap* top(bitmap * x, k)
 1: for mu ranging from 100 to 0 do
 2:     if k > 0 then
 3:         truncate(k, x[mu])
 4:     elsex[mu] = 0
 5: return x

   bitmap truncate(k, bitmapx)
 1: while xrun- > it < size(x) do
 2:     tmp ← getword(x, xrun.it)                    ▷ get new word
 3:     decode(xrun, *tmp)                           ▷ decode the current word
 4:     nWords+ = xrun.nWords                        ▷ update the word counter
 5:     if xrun.isFill and xrun.data = allones then
 6:         if setbitcount + 31 * xrun.nWords > k then
 7:             append trailing fills then a literal
 8:             set k = 0 and leave
 9:         setbitcount+ = 31 * xrun.nWords
10:     else
11:         if setbitcount + bitCount(xrun.data) > k then
12:             finds which bit exactly corresponds to k
13:             override trailing bit with 0
14:             set k = 0 and leave
15:         setbitcount+ = bitCount(xrun- > data)
16:     xrun.it ← xrun.it + 1                        ▷ points to next word
17: n- = setbitcount                                 ▷ remaining number of bits not found
```

For fuzzy sets requiring a fine level granularity, i.e., a high cardinality of membership degrees, the number of bits used to represent the membership degree on uncompressed arrays grows logarithmically. On compressed arrays, for fuzzy sets with at least one element per membership degree, no size difference will be noticed. Similarly, WAH bitmaps are well known to scale very well with high cardinality attributes as their size is bounded by the total number of elements and not the number of bitmaps.

Finally, the performance studies of the previously presented operators are directly applicable to fuzzy sets. For other operators, e.g., intersections defined using different t-norms, new performance studies are required. For WAH bitmaps, the computational time will be proportional to the number of logical bitwise operations required on the compressed bitmaps.

Algorithm 4.4 Top operation of a fuzzy song set represented by an array of WAH compressed bitmaps with membership degree ranging from 0 to 100

$bitmap*$ **reduce**$(bitmap * x, alpha)$
1: **for** mu ranging from $alpha - 1$ to 1 **do**
2: $x[mu] \leftarrow 0$
3: **return** x

4.9 Conclusions and Future Work

As music recommendation systems are becoming increasingly popular, new efficient tools able to manage large collections of musical attributes are urgently needed. Fuzzy sets prove to be well suited for addressing various problematic scenarios commonly encountered in recommendation systems. After defining fuzzy song sets and presenting an algebra to manipulate them, we demonstrate the usefulness of fuzzy song sets and their operators to handle various information management scenarios in the context of a music warehouse. For this purpose we create two multidimensional cubes: the Song Similarity Cube and the User Feedback Cube. Three data options, arrays, tables and WAH bitmaps, are envisioned for representing fuzzy song sets. We proceed by discussing the impact of these data structures on the storage space and operators performance.

With respect to storage, while arrays first show to be a very good choice from a theoretical point of view, they suffer from a significant overhead. Estimates taking into account DBMS overheads show that the differences between WAH bitmaps and arrays vanish as the number of elements grows. The different data organizations in WAH bitmaps and in arrays cause operators to behave very differently depending on the number of elements. Arrays are very efficient when the number of elements remains limited. However, due to frequent sorting operations, arrays behave poorly for larger sets. Requiring more complex management, bitmaps suffer from a higher starting overhead that is mostly visible when the number of elements is low. As the number of elements grows, operations on bitmap are faster than on arrays. In our experiment with the largest number of elements, the Union operator on WAH bitmaps is performed 5 times faster than on arrays, the speedup factor is 7 for the Top operator and 85 for the Reduce operator.

Future research directions include the development of methods for the transparent manipulation of arrays and bitmap and the automatic selection of a data structure option during the query plan optimization phase. Further research on how to improve the WAH compression performance by using a longer alignment without diminishing the compression ratio seems also promising, e.g., for 64 bits system architecture.

Chapter 5

Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps

Compressed bitmap indexes are increasingly used for efficiently querying very large and complex databases. The Word Aligned Hybrid (WAH) bitmap compression scheme is commonly recognized as the most efficient compression scheme in terms of CPU efficiency. However, WAH compressed bitmaps use a lot of storage space. This paper presents the Position List Word Aligned Hybrid (PLWAH) compression scheme that improves significantly over WAH compression by better utilizing the available bits and new CPU instructions. For typical bit distributions, PLWAH compressed bitmaps are often half the size of WAH bitmaps and, at the same time, offer an even better CPU efficiency. The results are verified by theoretical estimates and extensive experiments on large amounts of both synthetic and real-world data.

5.1 Introduction

Compressed bitmap indexes are increasingly used to support efficient querying of large and complex databases. Example applications areas include very large scientific databases and multimedia applications, where the datasets typically consist

of feature sets with high dimensionality. Here, compressed bitmap indexes enable efficient range queries over one dimension, as well as the combining several dimensions for multidimensional range queries. The present work was motivated by the need to perform (one- or multi-dimensional) range queries in large multimedia databases for music (Music Warehouses). Here, music snippets are analyzed on various cultural, acoustical, editorial, and physical aspects. The extracted features are high-dimensional and range over very wide intervals. However, the exact same characteristics apply to a wide range of applications within multimedia and scientific databases. While bitmap indexes are very efficient for queries, the size of the index increases dramatically for high-cardinality attributes without good compression schemes.

The first use of a bitmap index in a DBMS dates back to 1987 [64]. The index was made from uncompressed bitmaps and suffered from tremendous storage requirements proportional to the cardinality of the indexed attributes. As a result, it became too large to fit in memory and the performance deteriorated [94]. Since then, various approaches have been studied to improve bitmap indexes for high cardinality attributes. These improvements can be organized in two categories: extensions of the bitmap index structure and bitmap compression schemes. A deep review of bitmap index extensions, compression techniques, and technologies used in commercial DBMSes is presented in [85].

The *binning* technique partitions the values of the index attributes into ranges [78, 93]. Each bitmap captures a *range* of values rather than a single value. Binning techniques prove to be useful when the attribute values can be partitioned into sets. In [55], a binning technique using partitioning based on query patterns, their frequencies, and the data distribution is presented and further improves the index performance. *Bit-slicing* techniques rely on an ordered list of bitmaps [65]. If every value of an attribute can be represented using n bits, then the indexed attribute is represented with an ordered list of n bitmaps, where for example, the first bitmap represents the first bits of the values of the indexed attribute. Dedicated arithmetic has been developed to operate directly on the bitmaps in order to, for example, perform ranges queries [77]. The Attribute-Value-Decomposition (AVD) is another bit-slicing technique designed to encode both *range-encoded* and *equality-encoded* bitmap indexes [21]. Both lossy and lossless bitmap compression schemes have been applied to bitmap indexes. The Approximate Encoding, (AE), is an example of lossy bitmap compression scheme [9]. An AE compressed bitmap index returns false-positives but is guarantied not to return false-negatives. The accuracy of an AE compressed bitmap index ranges from 90% to 100%.

The Byte-aligned Bitmap Compression (BBC) [8] and the Word Aligned Hybrid (WAH) [92] are both lossless compression schemes based on run-length encoding. In run-length encoding, continuous sequences of bits are represented by one bit of

the same value and the length of the sequence. The WAH compression scheme is currently regarded the most CPU efficient scheme, and is faster than, e.g., BBC. The performance gain is due to the enforced alignment with the CPU word size. This yields more CPU friendly bitwise operations between bitmaps. However, WAH suffers from a significant storage overhead; an average of 60% storage overhead over BBC was reported [92]. More recently, for attributes with cardinalities up to 10,000, hybrid techniques based on combined Huffman run-length encoding have shown their superiority from a size point of view [84]. However, both storage and performance of such compressed bitmap indexes decrease for bigger cardinalities. Performing bitwise operations (OR/AND) on such bitmaps is very expensive since the bitmaps must first be de-compressed and later re-compressed. General-purpose data compression algorithms, such as PFOR and PFOR-DELTA [100], often offer very efficient decompression but are not well-suited for bitmap compression as they necessitate a preliminary decompression to operate on the values.

This paper improves upon existing work by offering a lossless compression technique that outperforms WAH, currently considered as the leading bitmap compression scheme for high cardinality attributes, from both a storage and a performance perspective. Other extensions to the bitmap index (bit-slicing, etc.) will thus also benefit from the proposed compression scheme to represent their underlying bitmaps.

Specifically, the paper presents Position List Word Aligned Hybrid (PLWAH), a new bitmap compression scheme. PLWAH is based on the observation that many of the bits used for the *run-length counter* in the WAH compression scheme are in fact never used, since runs never become long enough to need all the bits. Instead, these bits are used to hold a "position list" of the set/unset bits that follow a 0/1 run. This enables a significantly more efficient storage use than WAH. In fact, PLWAH compressed bitmaps are often only half the size of WAH compressed ones. As at the same time, PLWAH is faster than WAH, PLWAH provides "the best of both worlds." Specifically for uniformly distributed bitmaps, the hardest bitmaps to compress for compression schemes based on run length encoding, PLWAH uses half the space of WAH. On clustered bitmaps, PLWAH also uses significantly less storage than WAH. These results are shown in a detailed theoretical analysis of the two schemes. The paper also presents algorithms that perform efficient bitwise operations on PLWAH compressed bitmaps and analyzes their complexity. Again, the analysis shows that PLWAH is faster than WAH. The theoretical results are verified by extensive experimentation on both synthetic and real-world (music) data. The experiments confirm that PLWAH uses significantly less storage than WAH, while also outperforming WAH in terms of query speed. While developed in the specific context of Music Warehouses, the presented compression scheme is applicable to any kind of bitmaps indexes, thus making the contribution generally applicable.

The paper is structured as follows. Section 5.2 describes PLWAH compression scheme. In Section 5.3, we establish upper and lower bounds, provide size estimates for both uniformly distributed and clustered bitmaps, and discuss the impact of the size of the position list. The algorithms for performing bitwise operations on the compressed bitmaps are presented in Section 5.4, along with an analysis of the time complexity of the presented procedures. In Section 5.5, the theoretical size and time complexity estimates are verified through extensive experiments on both synthetic and real data sets. Finally, in Section 5.6, we conclude and present future research directions.

5.2 PLWAH Compression

The PLWAH compression is composed of four steps; they are explained with two examples to ensure clarity. Assume an uncompressed bitmap composed of 175 bits and a 32-bit CPU architecture. The PLWAH compression steps are detailed below and are illustrated in Figure 5.1.

Step 1 The uncompressed bitmap is divided into groups of equal size, corresponding to the word length of the CPU architecture minus one. In our example, the first five groups have a size of 31 bits. The last group is appended with 11 zeros to reach a size of 31 bits. We thus have six 31 bit long groups.

Step 2 Identical adjacent homogeneous groups, i.e., groups composed of either 31 set bits or 31 unset bits, are marked as candidates for a merge. In our example, the first group is a candidate for a merge since it is exclusively composed of homogeneous bits. Similarly, in the third and fourth groups, all bits are unset, so the two groups are candidates for a merge. The second, fifth, and sixth groups are not homogeneous and therefore are not candidates.

Step 3 An additional bit is appended to the groups at the position of their Most Significant bit (MSB). A set bit represents a group composed of *homogeneous* bits. Those 32 bit long words starting with their MSB set are referred to as *fill words*. Fill words can be of two types, *zero fill words* and *one fill words*; they are identified by their second MSB. Candidate groups for a merge are transformed into fill words. The last 25 Least Significant Bits (LSBs) are used to represent the number of merged groups that each fill word contains. In our example, the first group becomes a fill word. Similarly, the third and fourth groups become fill words with their counters set to two; this corresponds to the number of merged groups. An extra unset bit is added as MSB to *heterogeneous groups*. Encoded words having their MSB unset are referred to as *literal words*. In our example, the second, the fifth, and the sixth word are transformed into literal words; each

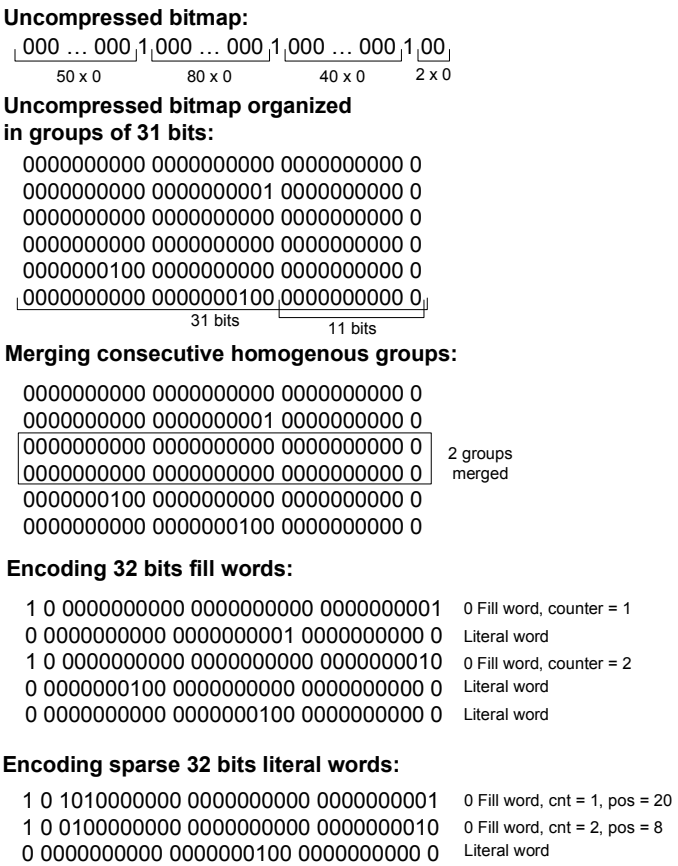


Figure 5.1: Example of PLWAH32 Bitmap Compression

starts with an unset bit. The first and second words are fill words, their MSBs are set.

Step 4 Literal words immediately following and "nearly identical"[1] to a fill word are identified. The positions of the heterogeneous bits are calculated and are placed in the preceding fill word. The unused bits located between the fill word type bit and the counter bits are used for this purpose. In our example, 25 bits are used for the counter, and 2 bits are used for representing the word type. We thus have 5 bits remaining, namely the 3rd to the 7th MSB. A heterogeneous bit requires 5 bits to that ensure all the possible bit positions in a literal word can be captured. In the example, the second and fourth words are literal words with

[1]The maximum number of bits differing between a literal word and a fill word to be considered as "nearly identical" will later be defined by a threshold parameter.

only one bit differing from their preceding fill word; they can be piggybacked. The position of the heterogeneous bit is placed in the position list of the fill word and the literal word is removed from the bitmap. The last word is treated as any other word; in the example, it cannot be piggybacked into its predecessor and is left as it is.

Now, assume an uncompressed bitmap composed of 262 bits and a 64-bit CPU architecture. As in the previous example, the PLWAH compression steps are detailed below and are illustrated in Figure 5.2.

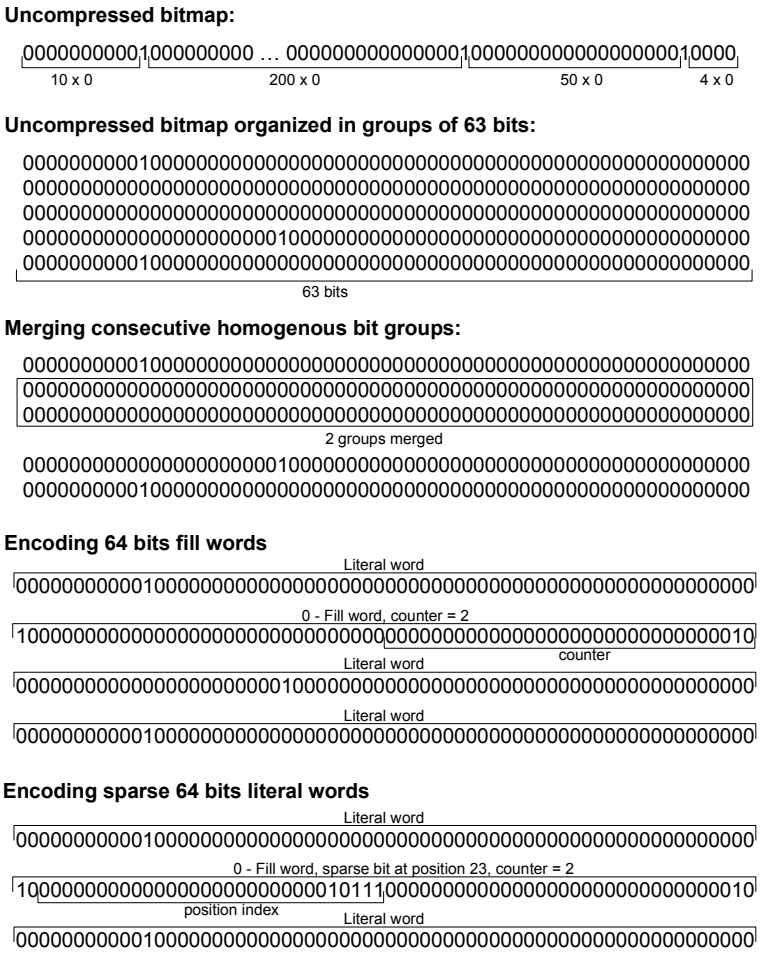


Figure 5.2: Example of PLWAH64 Bitmap Compression

Step 1 The first four groups have a size of 63 bits. The last group is appended with 48 zeros to reach a size of 63 bits. We thus have five 63 bit long groups.

Step 2 The first group is not a candidate for a merge since it is not exclusively composed of homogeneous bits. In the second and third groups, all bits are unset, so the two groups are candidates for a merge. The fourth and fifth groups are not homogeneous and therefore are not candidates.

Step 3 The second and third groups become a fill word. Its counter is set to two, this corresponds to the number of merged groups and not the total number of bits. The first, fourth and fifth word are transformed into literal words; each starts with an unset bit. The second word is created from merged groups and thus is stored using a fill word with its MSB set.

Step 4 Two bits are used for the representing the word type and 32 bits are used for the counter. We thus have 30 bits remaining, namely the 3rd to the 32nd MSB. A heterogeneous bit requires 6 bits to ensure all the possible bit positions in a literal word can be captured. Having 30 bits available and each position taking 6 bits, we thus have 5 positions for storing a literal word in its preceding fill word. In the example, the first word is a literal, so nothing is done. The second and third word are a fill word followed by a literal word with only one bit differing from the preceding fill word. The position of the heterogeneous bit is placed in the position list of the fill word and the literal word is removed from the bitmap. Nothing is done for the last word.

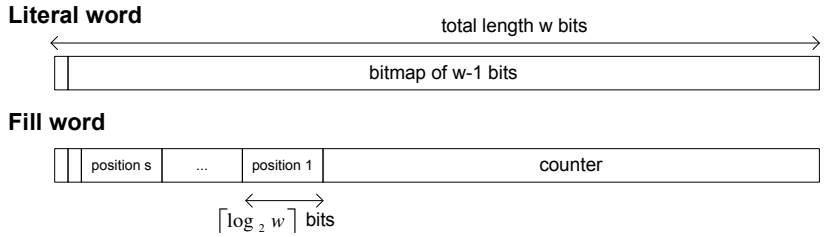


Figure 5.3: Structure of PLWAH Literal and Fill Words

As mentioned above, the PLWAH compressed words are of two kinds: fill words and literal words; their structures are presented in Figure 5.3. A literal word is identified by an unset MSB, followed by an uncompressed group. Fill words are identified by a set MSB, followed by a type bit to distinguish zero fills from one fills. The remaining bits are used for the optional storage of the list of positions of the heterogeneous bits in the tailing group, and for storing a counter that captures the total number of groups that the current fill word represents.

The position list can be empty in three cases: first, if the following word is a fill word of a different type; second, if the following word is a literal but is not "nearly identical" to the current fill word; and third, if the multiple fill words have to be repeated because the counter has reached its maximum value. An "empty" element in the position list is represented with all its value set to zero. The empty position list is represented with all position values set to zero.

5.3 PLWAH Size Estimates

In this section, we compare the space complexity of the WAH and PLWAH compression schemes. We discuss the influence of the word length and the size of the position list on the compression ratio for two different random distributions. We show that the PLWAH compression ratio is asymptotically twice better than the WAH.

Let w be the number of bits in the CPU word, so that $w = 32$ on a 32-bit CPU and $w = 64$ on a 64-bit CPU. Given an uncompressed bitmap as input, we described in Section 5.2 that the PLWAH compression scheme divides the uncompressed bitmap into groups of $(w - 1)$ bits, referred to as literal groups. For a bitmap with N bits, there are $\lfloor N/(w-1) \rfloor$ such groups plus an optional incomplete word of size $Nmod(w-1)$. Unset bits are appended to the incomplete word so that the bitmap is divided into $M = \lceil N/(w-1) \rceil$ groups each containing $(w - 1)$ bits. The total length in bits of the uncompressed bitmap with its tailing unset bits is $L = M(w-1)$. A literal word is created by prepending an unset bit to a group. When all groups of a bitmap are represented with literal words, the bitmap is said to be in its *uncompressed form*.

5.3.1 Compression Upper and Lower Bounds

The maximum compression is obtained when all words are fill words. This happens when all groups but the last are homogeneous and identical, and the last group can be represented using the list of positions of the preceding fill word. The upper bound is thus determined by the maximum number of groups a fill word can contain, plus one for the last literal group. Let s be the maximum number of heterogeneous bits a fill word can store in its list of positions. The size of the list is $s \log_2 w$ and the size of the counter is $w - 2 - s \log_2 w$. A single fill word can thus represent up to $\left(2^{w-2-s \log_2 w} + 1\right)$ groups of length $w - 1$. A very sparse bitmap composed of a long sequence of unset bits and ending with a set bit, $00000000...00001$, is an example of a bitmap were the maximum compression can be reached. Such bitmap can be represented with only one fill word.

All compression schemes have an overhead when representing incompressible bitmaps. For WAH and PLWAH, this overhead is one bit per word, so the compres-

sion ratio is $w/(w-1)$. A bitmap containing no homogeneous groups will not have any fill words and will be incompressible. The bitmap 01010101010101...0101010 is an example of an incompressible bitmap for both the WAH and PLWAH compression schemes.

The upper and lower bounds of the PLWAH compression ratio are respectively: $(2^{w-2-s\log_2 w}+1)(w-1)/w$ and $(w-1)/w$. As long as the upper bound is not reached, the worst PLWAH compression ratio is bounded by the WAH compression ratio. The bitmap compression ratio for particular data distributions is described in the following sections.

5.3.2 Compression of Sparse and Uniformly Distributed Bitmaps

Uniformly distributed bitmaps can be characterized with one parameter, namely their bit density d. The bit density is the fraction of bits that are set compared to the total number of bits. On a uniformly distributed bitmap of density d, the probability of having exactly k or less bits set in a sequence of w bits is given by the binomial distribution $P_u(k,w,d) = C_k^w d^k (1-d)^{w-k}$, where $C_k^w = \frac{w!}{k!(w-k)!}$ is the binomial coefficient and represents the total number of combinations, that k bits can be picked out of a set of w bits. The probability of having no bit set in a $w-1$ bit long word is $P_u(0,w-1,d) = (1-d)^{w-1}$. The probability of having all bits set is: $P_u(w-1,w-1,d) = d^{w-1}$. The probability of having two successive $w-1$ bit long words filled with unset bits is: $P_u(0,2w-2,d) = P_u(0,w-1,d)P_u(0,w-1,d) = (1-d)^{2(w-1)}$. Similarly, the probability of having two $w-1$ bit long words filled with set bits is: $d^{2(w-1)}$.

As shown by previous work [92], the number of words W in a compressed bitmap is $W = M - G$, where M is the total number of groups, and G is the number of pairs of blocks that can be collapsed. For the WAH scheme, G_{WAH} is the number of pairs of adjacent blocks containing only unset bits plus the number of adjacent blocks containing only set bits. The total number of adjacent blocks is $M-1$, and the expected value of G_{WAH} is $\overline{G}_{WAH} = (M-1)P_{col}$, where P_{col} is the probability of collapsing two adjacent blocks. The expected total number of words using WAH is:

$$\begin{aligned}
\overline{W}_{WAH} &= M - \overline{G}_{WAH} \\
&= M - (M-1)\Big[P_u\big(0,2(w-1),d\big) \\
&\qquad + P_u\big(2(w-1),2(w-1),d\big)\Big] \\
&= M\Big[1-(1-d)^{2(w-1)} - d^{2(w-1)}\Big] \\
&\qquad + (1-d)^{2(w-1)} + d^{2(w-1)}
\end{aligned} \tag{5.1}$$

Let L be the total length in bits of the uncompressed bitmap as defined in Sec-

tion 5.3.1. On a sparse bitmap, i.e., $d \to 0$, by applying a binomial decomposition, we have $1-(1-d)^{2(w-1)}-d^{2(w-1)} \to 2(w-1)d$. Considering large values of L (and thereby M) and small values of d, the expected number of words can therefore be approximated as follows:

$$\begin{aligned} \overline{W}_{WAH} &\approx M\Big[1-(1-2(w-1)d)\Big] \\ &= \frac{L}{w-1}2(w-1)d = 2Ld = 2h \end{aligned} \tag{5.2}$$

where h denotes the number of set bits. Using the definition of bit density, $h = dL$, the number of words in a sparse bitmap can be expressed in terms of set bits as shown in Equation 5.2. In such a sparse bitmap, all literal words contain only a single bit that is set, and each literal word is separated from the next by a fill word of zeros. On the average, two words are thus used for each bit that is set.

We now calculate the expected total number of words using the PLWAH compression scheme. The probability of having 0 to s set bits in a $w-1$ bit long group is: $\sum_{k=0}^{s} P_u(k, w-1, d)$. The probability of having a $w-1$ bit long group with $w-1$ unset bits followed by a $w-1$ bit long group with 0 to s set bits is: $P_u(0, w-1, d)\sum_{k=0}^{s} P_u(k, w-1, d)$. Similarly, the probability of having a $w-1$ bit long group with all its bits set followed by a group with 0 to s unset bits is: $P_u(w-1, w-1, d)\sum_{k=0}^{s} P_u(w-1-k, w-1, d)$. The expected total number of words is:

$$\begin{aligned} \overline{W}_{PLWAH} &= M-(M-1)\Big[P_u(0,w-1,d)\sum_{k=0}^{s}P_u(k,w-1,d) \\ &\quad + P_u(w-1,w-1,d)\sum_{k=0}^{s}P_u(w-1-k,w-1,d)\Big] \\ &= M-(M-1)\Big[(1-d)^{w-1}\sum_{k=0}^{s}C_k^{w-1}d^k(1-d)^{w-1-k} \\ &\quad + d^{w-1}\sum_{k=0}^{s}C_k^{w-1}d^{w-1-k}(1-d)^k\Big] \end{aligned} \tag{5.3}$$

For small values of d and large values of M, we can use a binomial decomposition. The expected number of words can then be approximated as follows.

$$\begin{aligned} \overline{W}_{PLWAH} &\approx M\Big[1-(1-d)^{2(w-1)}-(w-1)d(1-d)^{2w-3}\Big] \\ &\approx M(w-1)\Big[2d-d(1-(2w-3)d)\Big] \\ &\approx Ld = h \end{aligned} \tag{5.4}$$

Looking at the expected number of words relative to the expected number of set bits, we have:

$$\frac{\overline{W}_{WAH}}{h} \approx 2 \qquad \text{and} \qquad \frac{\overline{W}_{PLWAH}}{h} \approx 1 \tag{5.5}$$

The compressed size of a sparse bitmap is thus directly proportional to the number of bits set. In such a sparse bitmap, all words are fill words of unset bits with one position set in the position list of the fill word. The compression ratio of PLWAH is asymptotically twice the compression ratio of WAH for a uniformly distributed bitmap as the bit density goes down to zero.

Figure 5.4 shows the expected number of words per set bit for WAH and PLWAH depending on the bit density. The behavior of the curves for low densities is explained by the previous approximations detailed in Equation 5.5. For higher densities, the number of words per set bit drops linearly to the bit density: if the density is too high to have fill words, the bitmaps are filled with literal words, they have become incompressible and have reached their maximum size as explained by the lower bound of the compression ratio established in Section 5.3.1.

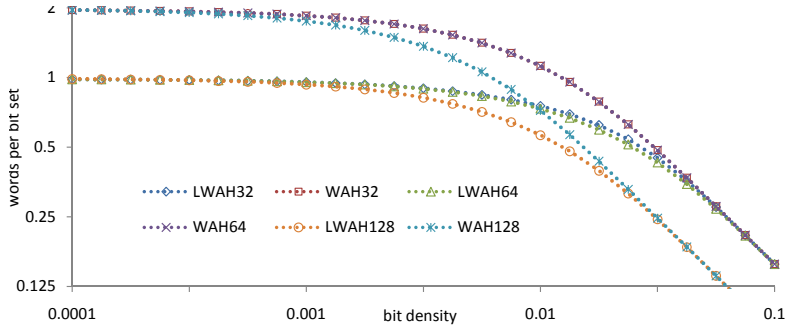


Figure 5.4: Word Count Estimates on Uniformly Distributed Bitmaps

5.3.3 Compression of Clustered Bitmaps

We now study the compression of a bitmap constructed by a random walk modeled by the following Markov process. A bit has a probability p to be set if its preceding bit is unset, and a probability q to be unset if its preceding bit is set. The bit density is: $d = (1-d)p + d(1-q) = p/(p+q)$. The clustered bitmap can thus be fully described by the density d and its clustering factor $f = 1/q$.

Let $P_c(k, w, d, f|b_0)$ denote the probability of having k set bits in a block of w bits when the bit preceding the block is b_0, the clustering factor is f and the bit density is d. It is straightforward to calculate $P_c(0, w, d, f|0) = (1-p)^w = (1 - d/f(1-d))^w$. Thus, the probability to have two consecutive groups exclusively composed of

set bits or unset bits is respectively $d(1-q)^{2(w-1)-1}$ and $(1-d)(1-p)^{2(w-1)-1}$. The expected number of compressed words in a WAH bitmap can thus be expressed as follows.

$$\begin{aligned}
\overline{W}_{WAH} &= M - \overline{G}_{WAH} \\
&= M - (M-1)\Big[(1-d)P_c(0, 2w-3, d, f|0) \\
&\qquad + dP_c(2w-3, 2w-3, d, f|1)\Big] \\
&= M - (M-1) \\
&\qquad \times \Big[(1-d)(1-p)^{2w-3} + d(1-q)^{2w-3}\Big] \\
&= M\Big[1 - (1-d)(1-p)^{2w-3} - d(1-q)^{2w-3}\Big] \\
&\qquad + (1-d)(1-p)^{2w-3} + d(1-q)^{2w-3}
\end{aligned} \tag{5.6}$$

For sparse bitmaps with $d << 1$, an upper bound for the number of words per set bit is to consider that almost all homogeneous groups are composed of unset bits. Furthermore, for $p = dq/(1-d) \to 0$ we can use a binomial decomposition; this is always true for $d << 1$ as $q <= 1$. Therefore, for large values of M, the expected number of compressed words is as follows.

$$\begin{aligned}
\overline{W}_{WAH} &\approx M\Big[1 - (1-d)\big(1 - dq/(1-d)\big)^{2w-3}\Big] \\
&\approx M\Big[1 - (1-d)\big(1 - (2w-3)dq/(1-d)\big)\Big] \\
&= \frac{L}{w-1}\Big[1 - 1 + d + (2w-3)dq\Big] \\
&= \frac{Ld}{w-1}\Big[1 + (2w-3)q\Big]
\end{aligned} \tag{5.7}$$

For bitmaps with a moderate clustering factor, $f < 5$, we have $\overline{W}_{WAH} \approx 2h/f$. Similarly, the expected number of words on a PLWAH bitmap is as follows.

$$\begin{aligned}
\overline{W}_{PLWAH} &= M - \overline{G}_{PLWAH} \\
&= M - (M-1) \\
&\qquad \times \Big[(1-d)(1-p)^{w-2} \sum_{k=0}^{s} P_c(k, w-1, d, f|0) \\
&\qquad + d(1-q)^{w-2} \sum_{k=0}^{s} P_c(w-1-k, w-1, d, f|1)\Big]
\end{aligned} \tag{5.8}$$

Unlike when $k = 0$, for $1 < k < w-1$, a recursive calculation is needed to

calculate $P_c(k, w, d, f|0)$. Nonetheless, on a clustered bitmap, a lower bound for $P_c(k, w, d, f|0)$ is the probability to have k set bits forming an uninterrupted sequence of set bits. The probability of having a group full of unset bits followed by a group with k set bits in sequence is: $(1-d)p(1-q)^{k-1}(1-p)^{2w-4-k}\big[(w-k-1)q+(1-p)\big]$. Therefore, an upper bound for the expected number of words in an PLWAH bitmap is to consider only uninterrupted sequences of set bits. The upper bound for $d << 1$ can be expressed as follows.

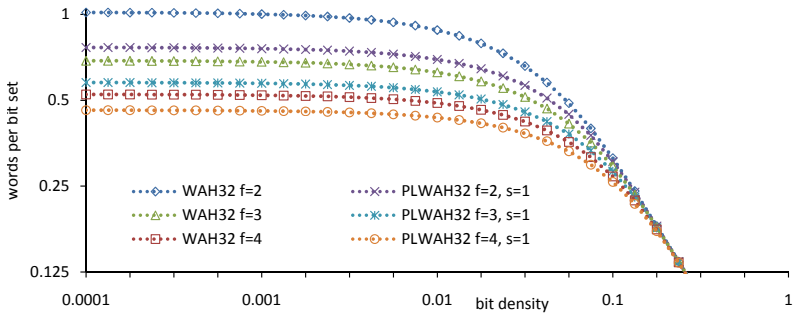
$$\begin{aligned}
&\lceil \overline{W}_{PLWAH} \rceil \\
&= M - (M-1) \\
&\quad \times \Big[(1-d)(1-p)^{2w-3} + d(1-q)^{2w-3} \\
&\qquad + \sum_{k=1}^{s} \Big((1-d)p(1-q)^{k-1}(1-p)^{2w-4-k} \\
&\qquad\quad \times \big[(w-k-1)q + (1-p)\big] \Big) \\
&\qquad + \sum_{k=1}^{s} \Big(dq(1-p)^{k-1}(1-q)^{2w-4-k} \\
&\qquad\quad \times \big[(w-k-1)p + (1-q)\big] \Big) \Big]
\end{aligned} \tag{5.9}$$

Furthermore, we can make the same assumption used in Equation 5.7; on a sparse bitmap with $d << 1$, almost all homogeneous groups are composed of unset bits. For $s = 1$, the expected number of words a PLWAH bitmap can be approximated as follows.

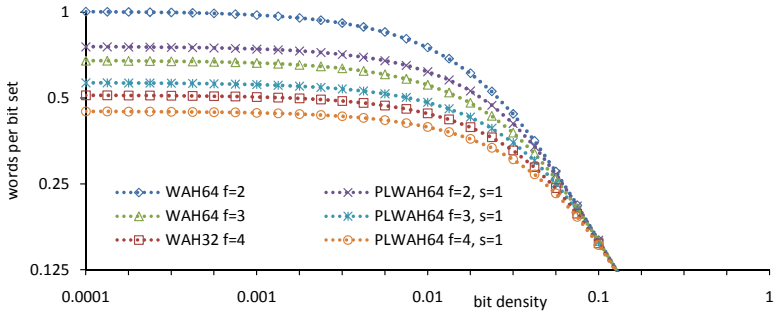
$$\begin{aligned}
W_{PLWAH} &\overset{s=1}{\approx} M\Big[1 - (1-d)(1-p)^{2w-3} \\
&\qquad - (1-d)p(1-p)^{2w-5}\big((w-2)q + (1-p)\big)\Big] \\
&\overset{f<5}{\approx} Ld(2q - q^2) = \left(2 - \frac{1}{f}\right)\frac{h}{f}
\end{aligned} \tag{5.10}$$

Equation 5.10 represents the size in words of a PLWAH bitmap where the maximum number of sparse bit positions a fill word can contain is constrained to one. The number of sparse bits that can be "piggybacked" into a fill word depends on w and the length of the counter as explained in Section 5.3.1.

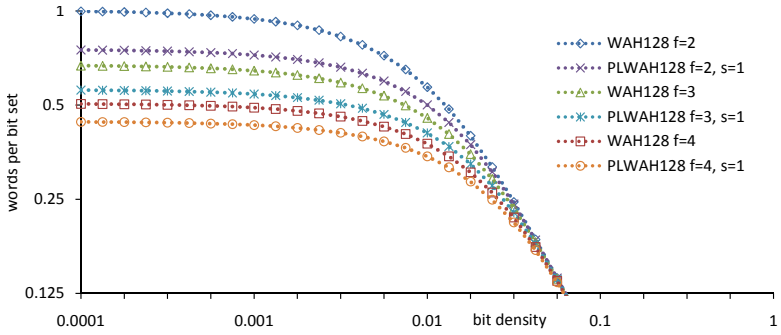
Figure 5.5 presents the average number of compressed words per set bit required by the WAH and PLWAH depending on the bit density for different values of the clustering factor. As established in Equation 5.10, the number of words in a PLWAH



(a) Clustered Bitmap on 32 Bit Long Words



(b) Clustered Bitmap on 64 Bit Long Words



(c) Clustered Bitmap on 128 Bit Long Words

Figure 5.5: Word Count Estimates on Clustered Bitmaps

bitmap decreases as q decreases, i.e., as the clustering factor increases. We also observe that PLWAH compression always requires less words per set bit than WAH compression. In Figure 5.5(a), the ratio between the number of words in a sparse

WAH bitmap and the corresponding PLWAH bitmap ranges from 1.32 to 1.13 for clustering factors varying from 2 to 4. Compression using 64 bit and 128 bit long words show identical ratios between the number of words in WAH and PLWAH compressed bitmaps. At higher bit densities, the bitmaps become incompressible and adopt a linear behavior. Finally, for very sparse bitmaps, the word length has little impact on the number of WAH and PLWAH words per set bit as shown by comparing the graphs from Figures 5.5(a), 5.5(b), and 5.5(c). However, as the bitmaps get denser, they become incompressible and the number of words per set bit is thus inversely proportional to the word length.

5.3.4 The Size of the Position List in Fill Words

The size of the position list depends on the number of bits available in a fill word. A fill word has to contain at least two bits, the remaining bits are shared between the position list and the counter. Increasing the size of the position list causes the total number of bits assigned to the counter to decrease. There is therefore a trade-off between the maximum number of groups that can be represented in a fill word, and the maximum number of heterogeneous bits in group that can be stored within its preceding fill word. However, we show that for most database applications, a position list does not impose practical limitations to the maximum number of indexed elements and that increasing the size of the position list improves the compression ratio significantly.

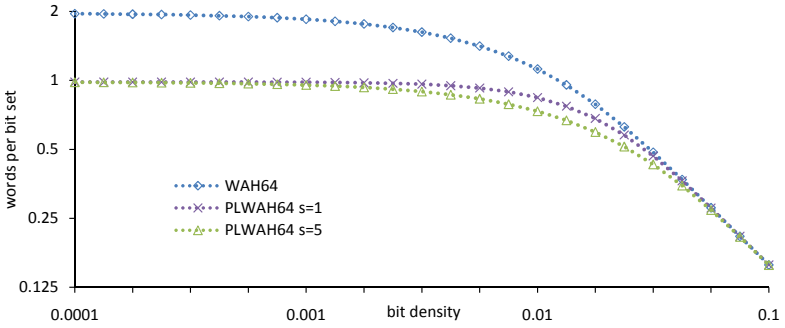
Each heterogeneous bit requires $\lceil \log_2 w \rceil$ bits to be stored in the position list. Let l be the number of bits taken by the position list and r the number of bits taken by the counter. The total number of bits in a fill word is : $w = 2 + l + r$. The maximum number of heterogeneous bits that can be stored in a fill word is: $s = \lfloor \frac{l}{\lceil \log_2 w \rceil} \rfloor = \lfloor \frac{w-2-r}{\lceil \log_2 w \rceil} \rfloor$.

In Equation 5.3, the cumulative distribution increases for increasing s. Similarly, all the terms in the $\sum^{s}$ in Equation 5.9 are positive. Thus, for both uniformly distributed and clustered bitmaps, W_{PLWAH} decreases when increasing s. Therefore, maximizing s minimizes W_{PLWAH}.

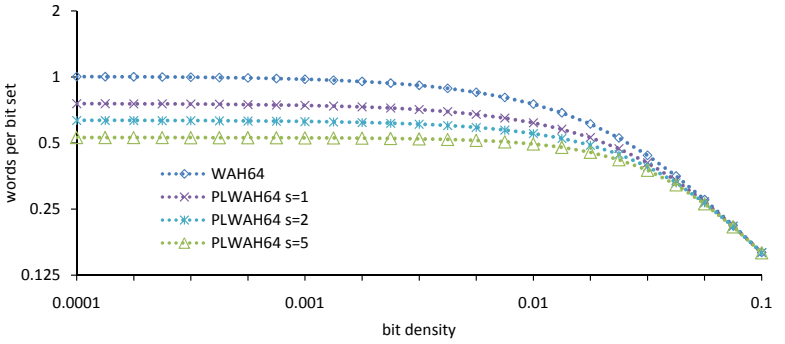
Figure 5.6 shows the effect of varying s for uniformly distributed and clustered bitmaps. There is very little benefit of increasing s on sparse uniformly distributed bitmaps as the probability of having multiple heterogeneous bits in a single group drops for low bit densities. However, bitmaps whose bit density is between the two linear zones contain fill words and literal words with a few set bits. For those bitmaps, increasing the value of s increases the probability of a fill word to be able to carry its following literal word. Thus, as s increases, the number of words per set bit decreases.

On a clustered bitmap, however, increasing s increases the compression ratio, in such a manner that for $\lim_{s \to w-1}$, $W_{PLWAH} = 0.5 * W_{WAH}$, i.e., all literal words with heterogeneous bits can be stored in their preceding fill word.

The length of an uncompressed bitmap in a bitmap index corresponds to the number of indexed elements. For $s = 1$, on a 32-bit CPU, PLWAH can represent $31 * 2^{25} > 1,000,000,000$ elements in a single fill word. For $s = 5$, on a 64-bit CPU, PLWAH can represent $63 * 2^{32} > 270,000,000,000$ elements in a single fill word. Almost any imaginable database application will thus only require a single fill word to capture homogeneous bit sequences.



(a) Uniformly distributed bitmap on 64 bit long words



(b) Clustered bitmap on 64 bit long words, f = 2

Figure 5.6: Word Count Estimates on Clustered Bitmaps

5.3.5 Compression of High Cardinality Attributes

In this section, we discuss the compression of a attribute whose *random value* follows a uniform distribution, and the compression of a *clustered attribute* whose probabil-

ities to change from one value to another depends on a Markov process as described in Section 5.3.3.

Let c be the cardinality of the random attribute. The attribute can thus be represented with c bitmaps. Since all the values have an equal probability of appearing, each bitmap is uniformly distributed with bit density: $d = 1/c$. For indexing attributes with high cardinality the total estimated size in bits of the c WAH compressed bitmaps can be approximated as follows.

$$\begin{aligned} \text{size}_{WAH} &\approx Lwc/(w-1) \\ &\quad \times \left[1 - (1-1/c)^{2(w-1)} - 1/c^{2(w-1)}\right] \end{aligned} \tag{5.11}$$

Similarly, the total estimated size in bits of the c PLWAH compressed bitmap is approximated as follows.

$$\begin{aligned} &\text{size}_{PLWAH} \\ &\approx Lwc/(w-1) \\ &\quad \times \Big[1 - \big(1 - 2(w-1)d + d^{2(w-1)} - d^{2(w-1)}\big) \\ &\qquad - (1-d)^{2(w-1)}wd - d^{2(w-1)}w(1-d)\Big] \end{aligned} \tag{5.12}$$

When c is large, $(1/c)^{2w-2} \to 0$ and $(1-1/c)^{2w-2} \to (1-(2w-2)/c)$. The total size of the c WAH compressed bitmaps that compose the bitmap index for a uniformly distributed random attribute has the following asymptotic formula.

$$\text{size}_{WAH} = 2Lw \text{ bits} = 2L \text{ words} \tag{5.13}$$

Similarly, the total estimated size of the c PLWAH compressed bitmaps is as follows.

$$\text{size}_{PLWAH} = Lw \text{ bits} = L \text{ words} \tag{5.14}$$

The same reasoning holds for a clustered attribute where the probabilities are allowed to depend on another value. One such example is a simple uniform c-state Markov process: from any state, the Markov process has the same transition probability q to other states, and it selects one of the $c-1$ states with an equal probability. The total expected size in bits of the WAH compressed bitmaps necessary to index a

clustered attribute is given by rewriting Equation 5.7 as follows.

$$\begin{aligned}
&\text{size}_{WAH} \\
&\approx \quad \frac{Lwc}{w-1}\Big(1-(1-1/c)\big(1-q/c(1-1/c)\big)^{2w-3}\Big) \\
&= \quad \frac{Lw}{w-1}\Big(1+(2w-3)q\Big) \\
&\overset{f<5}{\approx} \frac{2h}{f} \text{ words}
\end{aligned} \tag{5.15}$$

Using Equation 5.9, the total expected size of a corresponding bitmap index compressed with PLWAH is as follows.

$$\begin{aligned}
&\text{size}_{PLWAH} \\
&= cM - c(M-1) \\
&\quad \times \Big[(1-\frac{1}{c})(1-p)^{2w-3} + \frac{1}{c}(1-q)^{2w-3} \\
&\qquad + \sum_{k=1}^{s}\Big((1-\frac{1}{c})p(1-q)^{k-1}(1-p)^{2w-4-k} \\
&\qquad\quad \times \big[(w-k-1)q+(1-p)\big]\Big) \\
&\qquad + \sum_{k=1}^{s}\Big(\frac{q}{c}(1-p)^{k-1}(1-q)^{2w-4-k} \\
&\qquad\quad \times \big[(w-k-1)p+(1-q)\big]\Big)\Big] \\
&\overset{s=1,f<5}{\approx} L(2q-q^2) = \left(2-\frac{1}{f}\right)\frac{hw}{f} \text{ words}
\end{aligned} \tag{5.16}$$

The bit density in each bitmap of a bitmap index of a uniformly distributed attribute is inversely proportional to the attribute cardinality. Thus, for high cardinality attributes, the bit density is low for each bitmap of the bitmap index. The total size is proportional to the number of set bits. Similarly, for a high cardinality attribute following a clustered distribution, the bitmaps of the bitmap index are sparse. The total size is proportional to the number of set bits divided by the clustering factor, i.e., a clustered attribute take less storage space. For example, an attribute following a distribution with a clustering factor $f = 2$ takes half the storage of a uniformly distributed bitmap when compressed with PLWAH on 64 bit long words and a position list of size 5. Additionally, equations 5.13, 5.14, 5.15, and 5.16 show that for both WAH and PLWAH, and for all attribute distributions, the total size of the bitmap is proportional to the size of the alignment, i.e., the CPU word length. Thus,

for example, a bitmap index of an attribute following a uniform distribution takes the same size when compressed under PLWAH with a word alignment of 64 bits as when compressed with WAH with a word alignment of 32 bits.

5.4 Bitwise Operations

Logical bitwise operations are crucial operations for swiftly carrying out range queries and queries with multiple predicates using bitmap indexes. For this reason, we next examine the complexity of performing bitwise logical operations on PLWAH compressed bitmaps. In this section, we show that the time for completing an arbitrary logical operation between two PLWAH compressed bitmaps is proportional to the total size of the two compressed bitmaps. When the two operands are in an uncompressed form, the time required is thus constant. Finally, the time to perform a logical operation between a decompressed bitmap and a PLWAH compressed one is proportional to the size of the compressed bitmap.

5.4.1 Operations on Two Compressed Bitmaps

A bitwise operation on two compressed bitmaps is performed by sequentially scanning each input bitmap, compressed word by compressed word. The complexity of bitwise AND/OR operations are quite similar, but ANDs are somewhat faster since the resulting bitmap has less set bits than the operands (as opposed to ORs, where the opposite is true). Thus, for simplicity, we focus on the logical OR operation on two compressed bitmaps (the hardest of the two). However, the methods are very easily adapted to logical ANDs. Details on the OR implementation are found in Algorithm 5.4. Additionally, three auxiliary procedures are shown: *ReadWord*, *AppendFill*, and *AppendLit*.

The *ReadWord* procedure (Algorithm 5.1) is called each time a compressed word has to be decompressed. The decompression requires very few CPU cycles and has only one branch[2]. If the word is a fill word, it loads the corresponding fill group, adjusts the counter, and transforms the position list into a bitmap. Both the load of the data corresponding to the fill word type and the transformation from a position list to a bitmap are done by multiplication and bit shift operations and do not require any branching. If the word is a literal word, the data is loaded by masking out the MSB of the word.

The *AppendFill* procedure (Algorithm 5.2) generates a fill word corresponding to the provided fill word type and counter. It has one conditional branch.

Finally, the *AppendLit* procedure (Algorithm 5.3) transforms a literal group into a PLWAH encoded word. It has up to 4 conditional branches. It performs two inter-

[2]Reducing branching is important for keeping the CPU instruction cache full and for avoiding stalls.

Algorithm 5.1 Reads a compressed word and updates the run

ReadWord (Compressed word W)
1: **if** W is a fill word **then**
2: $data \leftarrow ((W >> 62)\&1) * all\_ones$
3: $nWords \leftarrow W\&counter\_mask$
4: $isFill \leftarrow true$
5: $isSparse \leftarrow (W\&position\_mask)! = 0$
6: $sparse \leftarrow$ bitmap constructed from the position list
7: **else**
8: $data \leftarrow W\&first\_bit\_unset$ ▷ MSB of W is unset
9: $nWords \leftarrow 1$
10: $isFill \leftarrow false$
11: $isSparse \leftarrow false$

Algorithm 5.2 Appends a fill word to a compressed bitmap

AppendFill(bitmap C, word $fill$, int $count$)
$last$ is the last word of the bitmap
1: **if** $last$ is same type as $fill$ and $last$ position list is empty **then**
2: $count$ is added to the counter of $last$
3: **else**
4: Append $fill$ to $bitmap$

esting operations, namely the count of the number of set bits and the transformation of a bitmap into a position list. Both operations can be efficiently performed using the new instructions at hand on recent CPU architectures. Counting the number of set bits can be performed directly using the *population count* instruction, part of the SSE4a instruction set available, for example, on the AMD "10h" [2] and the Intel "Core i7" [47] processor families. For older architectures, the most efficient alternative is probably to count the number of times the LSB of the bitmap can be removed before obtaining a bitmap with all bits unset. In practice, one can limit the count to a threshold corresponding to the maximum size that the position list can reach. Generating the list of the positions of the set bits in a bitmap is performed by locating the position of the LSB, removing the LSB, and repeating the process until all bits are unset. Many techniques to find the position of the LSB in a word exist in the literature [57]. However, the *bit scan forward* instruction, available for 32 and 64-bit words on modern CPU architectures, e.g., "Pentium 4", "AMD K8", and above, is by far the fastest way of tackling the task.

The *CCOR*, procedure (Algorithm 5.4) performs a bitwise OR on two compressed bitmaps. Its total execution time is dominated by the number of iterations through the main loop. Each iteration through the loop either consumes a fill word or a literal word from either one or both bitmap operands.

Algorithm 5.3 Appends a literal word to a compressed bitmap

```
    AppendLit(Compressed bitmap C, literal word L)
    last is the last word of C
 1: if L = 0 then
 2:     AppendFill(C, 0, 1)
 3: else if last is counter and has empty position list then
 4:     if last is a zero fill then
 5:         if L is sparse then                          ▷ popcount CPU instruction
 6:             Generate position list                   ▷ bitscan CPU instruction
 7:             Place position list into last
 8:         else
 9:             Append L to C
10:     else if L is sparse then                         ▷ popcount CPU instruction
11:         Generate position list                       ▷ bitscan CPU instruction
12:         Place position list into last
13:     else
14:         Append L to C
15: else
16:     Append L to C
```

Let W_x and W_y be respectively the number of words of each operand, and let M_x and M_y denote the number of words in their decompressed form, i.e., the number of groups. If each iteration consumes a word from both operands, it takes at least $\min(W_x; W_y)$ iterations. If each iteration consumes only one word from either operand, it may take $W_x + W_y$ iterations. Additionally, since each iteration produces at least one group and the result contains at most $\min(M_x, M_y)$ groups, the main loop requires at most $min(M_x, M_y)$ iterations[3]. The number of iterations through the loop I satisfies the following conditions:

$$\min(W_x; W_y) < I < \min\left((W_x + W_y), \min(M_x, M_y)\right) \tag{5.17}$$

When the operands are two sparse bitmaps, each word is a fill word with a non-empty position list. In that case, each operation only consumes one word from one of the operands. Therefore, it takes $W_x + W_y$ iterations to complete the bitwise OR, as each loop iteration executes *AppendFill* and *AppendLit* once. Compared to a WAH bitmap where each set bit requires on average two words, half the number of loop iterations are thus required. In the case of WAH bitmaps, every two iterations, the *AppendFill* procedure is called and followed by, in the next iteration, a call to the *AppendLit* procedure. Thus, the total number of calls to the *AppendFill* and *AppendLit* procedures to perform an OR operation on two PLWAH bitmaps and two WAH bitmaps are equal.

[3] Our implementation allows to perform bitwise operations using bitmaps of different uncompressed sizes. In the case of an OR, the remaining part of the longest bitmap is appended to the result.

Algorithm 5.4 Performs a bitwise OR on 2 compressed bitmaps

CCOR(Compressed bitmap X, Compressed bitmap Y)
$xrun$ holds the current run of X
$yrun$ holds the current run of Y
Z is the resulting compressed bitmap
1: Allocate memory for Z
2: **while** $xrun.it < size(X)$ and $yrun.it < size(Y)$ **do**
3: **if** $xrun.nWords = 0$ **then** $Readword(X, xrun.it)$
4: **if** $yrun.nWords = 0$ **then** $Readword(Y, yrun.it)$
5: **if** $xrun.isFill$ and $yrun.isFill$ **then**
6: $nWords \leftarrow min(xrun.nWords, yrun.nWords)$
7: $AppendFill(Z, xrun.data|yrun.data, nWords)$
8: decrease $xrun.nWords$ by $nWords$
9: decrease $yrun.nWords$ by $nWords$
10: **else**
11: $AppendLit(Z, xrun.data|yrun.data)$
12: **if** ($xrun.nWords = 0$ and $xrun.isSparse$)
or ($yrun.nWords = 0$ and $yrun.isSparse$) **then**
13: **if** $xrun.nWords = 0$ and $xrun.isSparse$ **then**
14: $xrun.data \leftarrow xrun.sparse$
15: $xrun.isFill \leftarrow false$
16: $xrun.isSparse \leftarrow false$
17: $xrun.nWords \leftarrow 1$
18: **if** $yrun.nWords = 0$ and $yrun.isSparse$ **then**
19: $yrun.data \leftarrow yrun.sparse$
20: $yrun.isFill \leftarrow false$
21: $yrun.isSparse \leftarrow false$
22: $yrun.nWords \leftarrow 1$
23: **if** $xrun.nWords > 0$ and $yrun.nWords > 0$ **then**
24: $AppendLit(Z, xrun.data|yrun.data)$
25: Decrement $xrun.nWord$
26: Decrement $yrun.nWord$
27: **if** $xrun.nWords = 0$ and $xrun.isSparse$ **then**
28: $xrun.data \leftarrow xrun.sparse$
29: $xrun.isFill \leftarrow false$
30: $xrun.isSparse \leftarrow false$
31: $xrun.nWords \leftarrow 1$
32: **else if** $yrun.nWords = 0$ and $yrun.isSparse$ **then**
33: $yrun.data \leftarrow yrun.sparse$
34: $yrun.isFill \leftarrow false$
35: $yrun.isSparse \leftarrow false$
36: $yrun.nWords \leftarrow 1$
37: **if** $xrun.nWords = 0$ **then** Increment $xrun.it$
38: **if** $yrun.nWords = 0$ **then** Increment $yrun.it$
39: Append the remaining of the longest bitmap to Z
40: **return** Z

The time complexity for performing a logical OR on two compressed bitmaps, T_{CC}, mainly depends on four terms: the time to perform one memory allocation T_a, the time to perform I decompressions T_d, the time to perform I *AppendFill* T_f, and the time to perform I *AppendLit* T_l. Let C_d, C_f, and C_l be respectively the time to invoke each of these operations. As established in Equation 5.17, $(W_x + W_y)$ is an upper bound for I. In common memory management libraries, the time for memory allocation *a*nd initialization is less than proportional to the size of the memory allocated [39]. Let C_a be the time to allocate one word, we thus have $T_a < C_a(W_x + W_y)$. An upper bound for the total time complexity is as follows.

$$\begin{aligned} T_{CC} &= T_a + T_l + T_f + T_d \\ &< (C_a + C_l + C_f + C_d)\,(W_x + W_y) \end{aligned} \tag{5.18}$$

The complexity of performing an OR operation on two compressed bitmap is $O(W_x + W_y)$.

5.4.2 In-place Operations

The most time expensive operations to perform a logical OR between two compressed bitmaps are in decreasing order: the memory allocation, the addition of a literal word, the addition of a fill word, and the decompression of a word. If an OR operation is executed on many bitmaps, the memory management dominates the total execution time. However, logical bitmap operations such as OR or AND are frequently used on a large number of sparse bitmaps in order, for example, to answer range queries or combine predicates. A common approach to reduce the memory management cost is to use an "in-place" operator that recycles the memory allocated for one of the operands to store the result, thus eliminating expensive memory allocation. For example, instead of allocating memory for z, and performing $z = x$ OR y, the "in-place OR" does $x = x$ OR y.

The PLWAH in-place OR takes one uncompressed and a compressed bitmap as operand. Using an uncompressed operand ensures that there is never more input than output words, so that no result word is overwriting a future input word. The uncompressed bitmap can thus be used both as input and output. In addition to avoiding repeated allocation of new memory for the intermediate results, it also removes the need to compress temporary results. The time complexity is thus mainly dependent on the time for initially allocating memory for the storage of the uncompressed bitmap and to perform I decompressions. The in-place OR operation is performed by the UCOR function whose details are supplied in Algorithm 5.5. UCOR is composed of a main loop that iterates through each word of the compressed operand.

Let x and y be respectively the uncompressed and compressed operand bitmaps. The number of words in x is M_x, the number of compressed words in y is W_y. Let

Algorithm 5.5 Performs a bitwise *OR* on an uncompressed bitmap and a compressed bitmap

```
    UCOR(Uncompressed bitmap U, Compressed bitmap C)
 1: for all words in C do
 2:     if run.data = 0 then
 3:         move forward of run.nWords words in U
 4:     else
 5:         while nWords do
 6:             nWords ← nWords − 1
 7:             Next word in U ← |run.data                ▷ "|" is a bitwise OR
 8:     if run.isSparse then
 9:         Next word in U ← |run.sparse                 ▷ "|" is a bitwise OR
```

I be the number of iterations in the main loop. Each iteration of the loop treats one compressed word, therefore there are $I = W_y$ iterations. The time to allocate memory for storing x, T_x, is at least proportional to M_x. Thus, $T_a < C_a M_x$. Let C_i be the time to process one iteration, the time to process the whole loop is proportional to $C_i I$. An upper bound for the total time spent to perform an in-place OR T_{UC}, is therefore:

$$T_{UC} = C_a M_x + C_i W_y \tag{5.19}$$

The UCOR has three conditional branches, including one conditional branch for decompressing a word. In comparison, the in-place OR algorithm has a total of two conditional branches per compressed word, but the number of compressed words in a WAH bitmap is always higher than the number of words found in the corresponding PLWAH bitmap. As explained in Section 5.3, the number of words in a PLWAH bitmap tends to be half the number of words in a WAH bitmap. Executing the UCOR thus represents a net performance gain for sparse bitmaps.

Equations 5.18 and 5.19 show that the fastest procedure depends on the size of the uncompressed bitmap and the size of the compressed bitmap, i.e., the compression ratio. Let r be the number of logical bitwise OR operations performed to answer a range query. The total time to perform the logical OR operations on compressed bitmaps to answer a range query is as follows.

$$\begin{aligned} T_{CC,r} &\approx r\,(C_a + C_l + C_f + C_d)\,(W_x + W_y) \\ T_{UC,r} &\approx C_a M_x + r(C_i W_y) \end{aligned} \tag{5.20}$$

The in-place benefits from a lower number of iterations in its main loop. In addition, the complexity of the main loop is significantly reduced compared to the OR operation on two compressed bitmaps. However, the in-place OR procedure suffers from a high startup cost due to the initial memory allocation corresponding to the size of an uncompressed bitmap.

Since, in a bitmap index, the size of the uncompressed bitmap is proportional to the size of the dataset, the fastest procedure to perform a range query depends on the size of the dataset and the range of the query.

5.5 Experiments

In this section, we present the timing results from a Dell Dimension 9300 equipped with a Intel "Core 2 6700 2.66GHz" CPU running FreeBSD 7.1 64-bit OS, which relies on the "jemalloc" [39] memory allocation library. All the presented experiments are performed within a PostgreSQL 8.3 DBMS. The implementation is in C and runs as a loadable module for PostgreSQL, thus making the code usable by a wide audience.

The experiments are conducted on both synthetic and real data sets. The generated data is composed of a set of *(key, attribute)* pairs. The attribute follows either a uniform distribution, or a clustered distribution as presented in Section 5.3.5. For each, a comparative study of the influence of the distribution parameters on the size and performance of the WAH and PLWAH indexes is conducted. The present study shows that PLWAH compression is superior to WAH compression. Furthermore, PLWAH performs "in-place OR" operations faster than WAH. For long range queries, the speed-up raises to 20%. As the performance-wise superiority of WAH over BBC is already shown in previous work [92], we only compare PLWAH with WAH.

Finally, experiments are conducted on a real data set composed of 15,000,000 music segments. Each music segment is described using 15 attributes capturing the results of music feature extraction algorithms. Each attribute has a cardinality of 100,000. The index keys are composed of the attribute identifier and the attribute value. The index keys are in turn indexed using the B-tree index available in PostgreSQL. The index thus has $15 \times 100,000$ entries and compressed bitmaps.

5.5.1 Bitmap Index Size

The sizes of the bitmap indexes for the various data distributions discussed earlier are presented in Figures 5.7, 5.8, 5.9, and 5.10 . Figure 5.7 confirms that for uniformly distributed high cardinality attributes, the size of PLWAH tends to be half the size of WAH for a given word length, in accordance with the estimates in Equations 5.13 and 5.14. Hence, the curves of PLWAH64 and WAH32 overlap (the WAH32 curve is actually hidden below the PLWAH64 curve).

The size of bitmap indexes on clustered attributes are shown for different clustering factors in Figures 5.8, 5.9, and 5.10. Again, PLWAH outperforms WAH on the compression ratio. PLWAH64 provides significant benefits due to its longer position

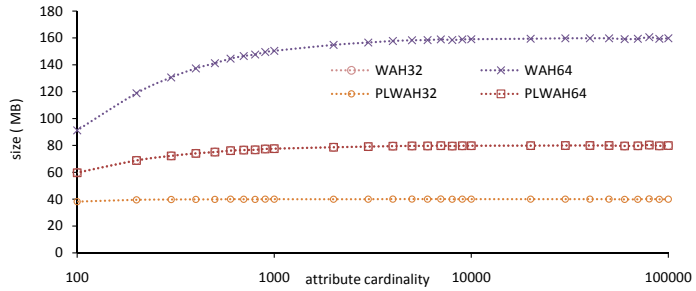


Figure 5.7: Size of Bitmap Index for a Uniformly Distributed Attribute
(Indexed Elements: 10,000,000)

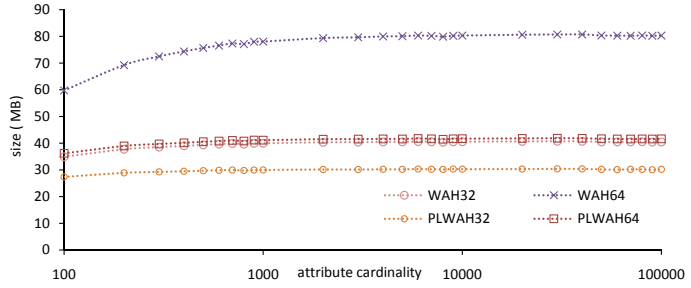


Figure 5.8: Size of Bitmap Index for a Clustered Attribute, $f = 2$
(Indexed Elements: 10,000,000)

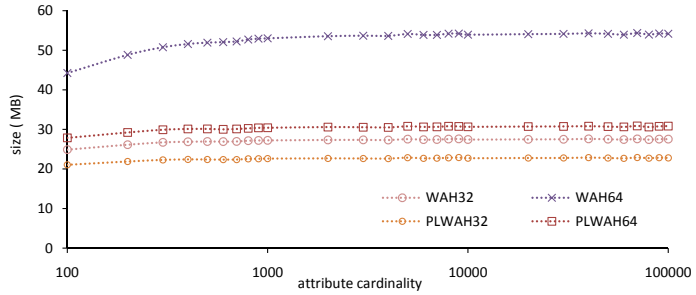


Figure 5.9: Size of Bitmap Index for a Clustered Attribute, $f = 3$
(Indexed Elements: 10,000,000)

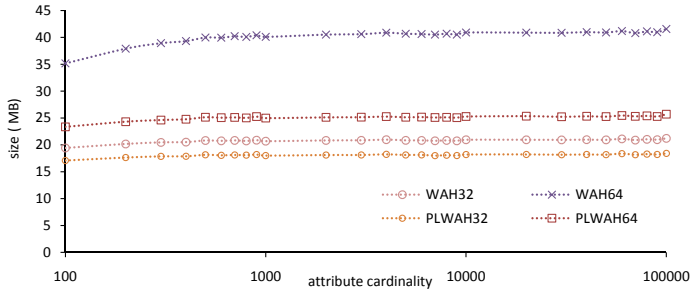


Figure 5.10: Size of Bitmap Index for a Clustered Attribute, $f = 4$
(Indexed Elements: 10,000,000)

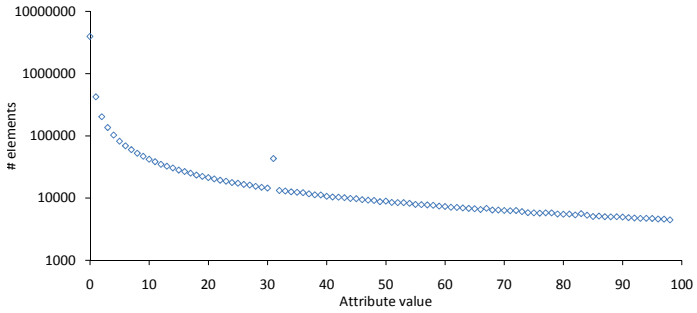
list. For short position lists, a higher clustering factor decreases the probability for PLWAH to "piggyback" the next literal word in the current fill word.

Data set	PLWAH64	WAH64	PLWAH32	WAH32
Uniform	86 MB	177 MB	43 MB	86 MB
Clustered, $f = 2$	48 MB	88 MB	36 MB	46 MB
Clustered, $f = 3$	37 MB	60 MB	28 MB	33 MB
Clustered, $f = 4$	31 MB	47 MB	24 MB	27 MB
15 music att.	926 MB	1617 MB	565 MB	926 MB

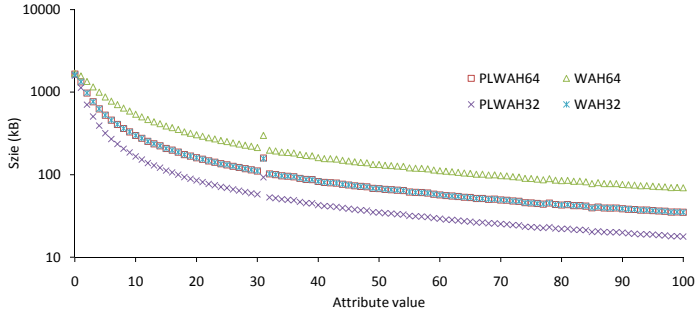
Table 5.1: The size of the bitmap indexes
for a common attribute cardinality of 100,000

The histograms of the music data set vary greatly from value to value. As shown in Figure 5.11(a), low values tend to be very frequent, e.g., 30% of the elements have 0 as value for the first extracted feature. For frequent values, the four compression schemes converge; the bitmaps are too dense to be compressed and the bitmaps are mainly composed of literal words. For less frequent values, the bitmaps can be compressed. PLWAH32 tends to be half the size of WAH32, PLWAH64 half the size of WAH64 and very similar to the size of WAH32. Figure 5.11(b) presents those results. For higher attribute values of music features, the histograms are more chaotic; nonetheless, the ratio between the compression schemes remains as previously observed. A comparison of the total index size between each compression scheme for the reviewed data distribution is presented in Table 5.1.

Finally, evidences of the influence of the position list size are uncovered in Figures 5.13(a) and 5.13(b). For uniformly distributed attributes, the probabilities of having multiple set bits in a single literal word are very low and the position list tends to contain only one set bit position. The gain obtained by increasing the size



(a) Histogram of the 100 Smallest Values of the First Music Feature Attribute



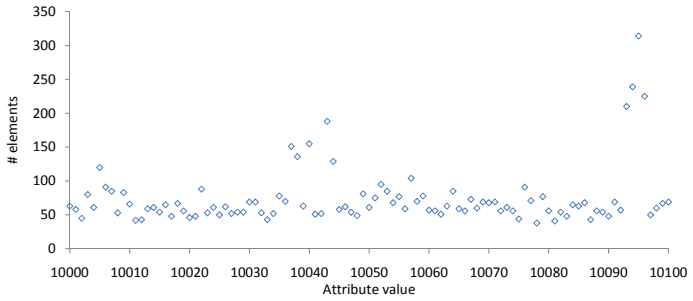
(b) Size of Bitmaps Corresponding to Each of the 100 Smallest Values of the First Music Feature Attribute

Figure 5.11: Histogram and Bitmap Sizes of the Lowest Values of the First Music Feature Attribute

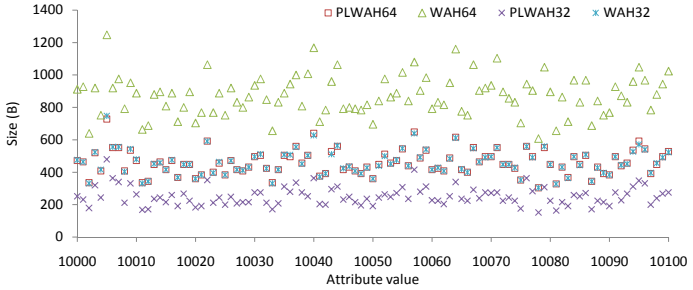
of the position list is thus marginal. The position list size proves to have an important impact on the compression ratio for clustered attributes. Indeed, for clustered attributes, the probability of having more than one set bit in a literal word increases with the clustering factor. Extending the position list allows more literal words to be encoded within their preceding fill word.

5.5.2 Performance Study

In the following, we study the performance of the OR operator for answering range queries. A similar study can be conducted using other bitwise operators. However, (1) OR operations are more complicated to handle for the compression scheme as the bitmaps become denser, which is not always the case, e.g., with the AND operator; and (2) long series of OR operations are often required to perform range queries, while the number of AND operations, for example required to treat a multi-



(a) Histogram for Values from 10,000 to 10,100 of the First Music Feature Attribute



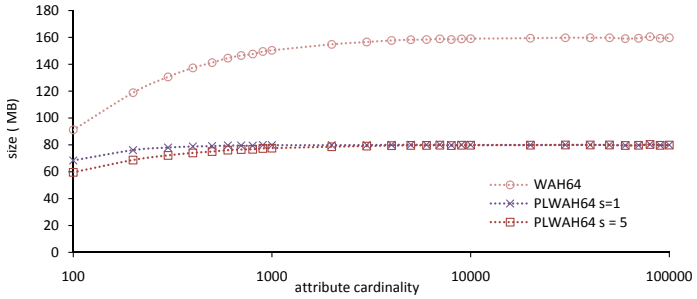
(b) Size of Bitmaps Corresponding to Values 10,000 to 10,100 of the First Music Feature Attribute

Figure 5.12: Histogram and Bitmap Sizes of Selected Values of the First Music Feature Attribute

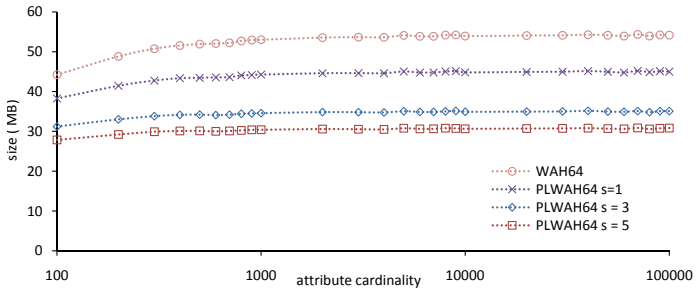
dimensional range query, tends to be much smaller. Range queries are generated by choosing a range of a given size randomly within the interval $[0, 100, 000]$. In the experiments, range queries are on one attribute at a time, i.e., we do not consider multi-dimensional range queries.

The performance of range queries on random intervals comprised of 10 values is shown in Figures 5.14(a) and 5.14(b). When the attribute cardinality increases, the compression ratio increases, and the number of words per bitmap decreases. Hence, the CPU time required to process a constant number of bitmaps decreases.

The total CPU time of performing range queries depending on the range is presented in Figures 5.15(a) and 5.15(b) for each of the two OR procedures. The behavior of each curve is discussed next. In a bitmap index, each possible value of the indexed attribute is represented with a bitmap; exactly one bit is set at any given position across all the bitmaps. Thus, when performing an OR operation, the number of set bits in the resulting bitmap is the total number of set bits in each operand bitmap. On a uniformly distributed attribute, the bit density per bitmap is constant



(a) Size of Bitmap Index for a Uniformly Distributed Attribute

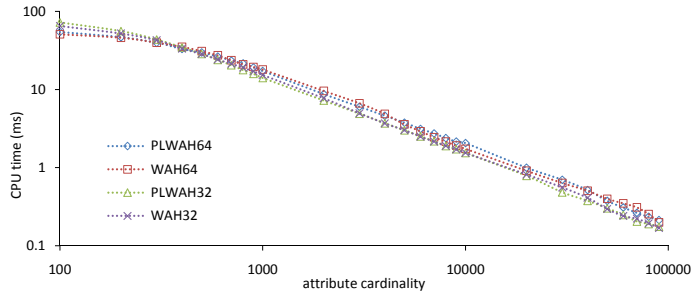


(b) Size of Bitmap Index for a Clustered Attribute, $f = 3$

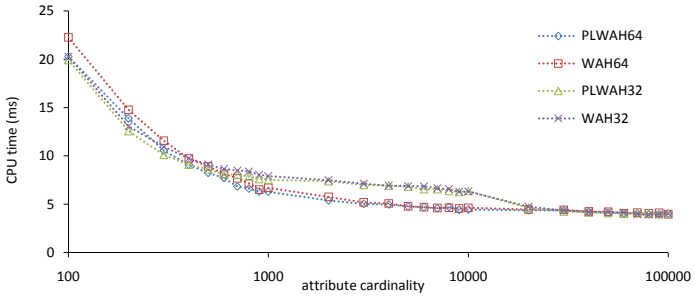
Figure 5.13: Size Comparison between PLWAH Bitmap Indexes with Different Position List Sizes (Indexed Elements: 10,000,000)

and since all bitmaps have the same uncompressed length, the expected number of set bits in each bitmap h is constant. Each OR operation thus increases the number of set bits approximately by h. After i OR operations, the result has ih set bits. Therefore, as explained by Equations 5.13 and 5.14, on very sparse bitmaps the lengths of the resulting WAH and PLWAH bitmaps after i OR operations are, respectively, $2ih$ and ih words long. Since, on sparse PLWAH bitmaps, each loop iteration produces one fill word with a non-empty position list, the total number of iteration to produce the i^{th} result is ih. Similarly, on WAH bitmaps, each loop iteration produces one word, thus, in total, $2ih$ iterations are required to produce the i^{th} result. To answer a range query, $r - 1$ OR operations are performed, thus $\sum_i^r ih = hr(r-1)/2$ and $\sum_i^r ih = hr(r-1)$ loop iterations are run. The complexity of performing a range query using the CCOR procedures on two compressed bitmaps is quadratic in the range.

The CCOR procedure offers good response time for queries on short ranges as show in Figure 5.15(a). Experiments show no significant performance drop for ranges covering less than 20 attribute values. In fact, PLWAH32 and WAH32 performances



(a) CPU Time to Perform CCOR on 10 Bitmaps

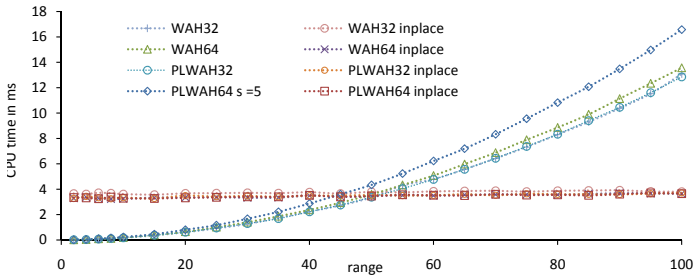


(b) CPU Time to Perform UCOR on 10 Bitmaps

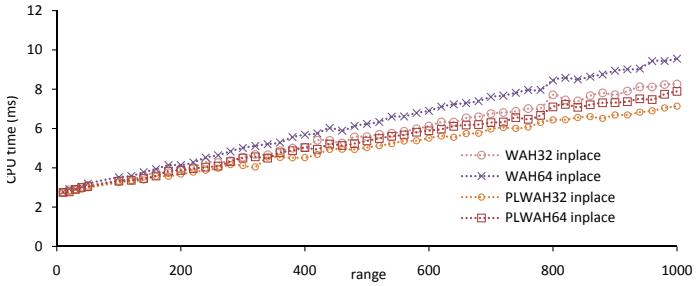
Figure 5.14: Performance Impact of the Attribute Cardinality (10,000,000 Elements, Uniformly Distributed Attribute)

are so similar that their performance curves overlap. For larger ranges, the performance of both WAH and PLWAH drops. For those large ranges, the WAH performance tends to be slightly better than the PLWAH due to the more complex AppendLit procedure present in PLWAH. However, these observations for larger range queries are not relevant, as for both algorithms, the "in-place OR" procedures prove to be much faster.

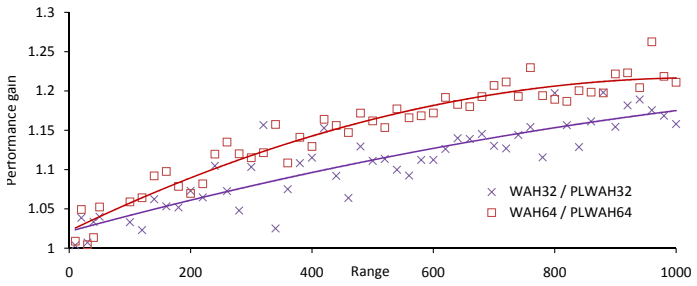
As illustrated in Figure 5.15(b), large range queries are better handled by the UCOR procedure, whose time complexity is linear in the size of the range. However, the UCOR procedure suffers from a high start-up cost mainly due to the initial memory allocation. For short range queries, the CCOR procedure is thus preferred. The maximum size of the position list does not change the complexity of the decompression or the management of a sparse literal; for clarity purposes, Figure 5.15(b) only shows PLWAH64 with a position list of size 5, and PLWAH32 with a position list of size 1. Both the WAH and PLWAH complexities depend on the size of the compressed operand bitmap. The small additional complexity of the decompression and the accounting of sparse literal words do not handicap the performance of PLWAH.



(a) Total Query Time using CCOR



(b) Total Query Time using UCOR

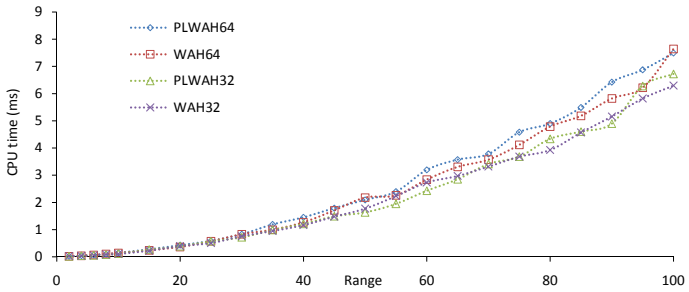


(c) Query Time Ratio WAH / PLWAH

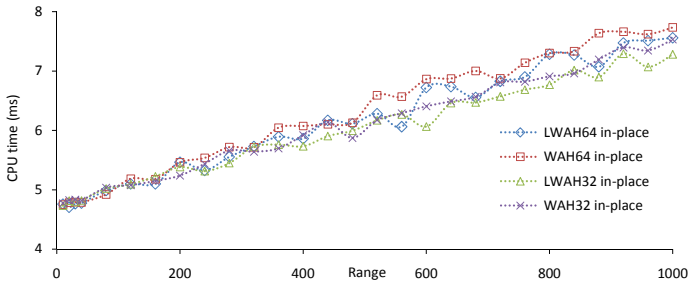
Figure 5.15: Performance on Uniformly Distributed Attribute (Elements: 10,000,000, Cardinality: 100,000)

On the contrary, PLWAH is more efficient for executing the UCOR procedure as only half the number of loop iterations are required. As shown in Figure 5.15(c), the ratios between the CPU time of WAH and PLWAH increase up to 20% as the range increases. The chaotic start is due to the high influence of the initial memory allocation. Better memory management, planned as future work, would further improve the ratio.

Figures 5.16(a) and 5.16(b) show similar results on the music data set. The noteworthy higher starting cost of the in-place OR operator on the music data set is due to the larger number of elements: the uncompressed bitmaps are 50% longer due to the 50% increase in the number of elements. However, the flatter slope after the startup overhead is due to the smaller size of the compressed bitmaps; bitmaps of uniformly distributed attributes are the hardest to compress, hence better compression ratios are obtained on the music data. For the same reason, a flatter slope is also noticeable for the OR on compressed bitmaps.



(a) Total CPU Time using CCOR on the Music Attributes



(b) Total CPU Time using UCOR on the Music Attributes

Figure 5.16: Performance Comparison between the WAH and PLWAH OR Operators on the Music Attributes

Finally, the impact of the position list size on the performance of the CCOR operator is shown in Figure 5.17. The computation of the position list is directly proportional to its maximum size. As the size of the position list increases, a small overhead can be observed for long range queries. However, as explained previously, better performance is achieved using the UCOR algorithm. Thus, the important results correspond to queries on small ranges, for which no significant overhead is observed. The decompression time is independent of the size of the position list.

Hence, the performance of the UCOR operation is not directly influenced by of the size of the position list.

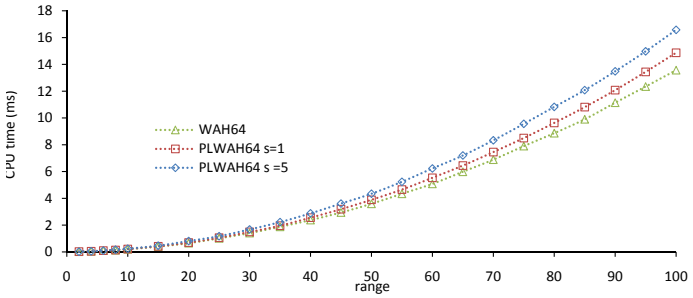


Figure 5.17: Performance Impact of the Size of the Position List on a Uniformly Distributed Attribute

In summary, the experiments conform with the analytical estimates in Equations 5.13 and 5.14; a uniformly distributed attribute indexed with PLWAH bitmaps takes half the size it would require using WAH. Furthermore, we have measured the performance impact of three factors, namely the attribute cardinality, the range and the size of the position list. For long range queries PLWAH shows a significant performance improvement. For short range queries, PLWAH and WAH efficiency is equivalent.

5.6 Conclusions and Future Work

In this paper, we present Position List Word Aligned Hybrid (PLWAH), a bitmap compression scheme that compresses sparse bitmaps better and answers queries on long ranges faster than WAH, which was so far recognized as the most efficient bitmap compression scheme for high cardinality attributes. The results are verified through detailed analytical and experimental approaches. The storage gain essentially varies depending on the following parameters: the size of the data set, the attribute distribution, the attribute cardinality, and the word length. For uniformly distributed high cardinality attributes, we both prove and observe that the compression ratio is twice as good as for WAH. For real data, the size of PLWAH compressed bitmaps varies between 57% and 61% of the size of WAH compressed bitmaps. In terms of performance, PLWAH and WAH are comparable for short range queries. However, for long range queries, PLWAH is up to 20% faster than WAH, depending on the data distribution.

Future work encompasses studying the performance impact of PLWAH on complementary bitmap indexing strategies, and collecting empirical storage and perfor-

mance results from real data sets and more concrete CPU types. Promising research directions include a dedicated primary index tool that builds aggregated bitmaps based on query patterns and frequencies, and data distribution. The primary index would then be able to select the most efficient aggregation path, the required bitwise operations, and the type of bitwise algorithm to use.

Appendix

5.A No Trailing Fill and No Active Word

In a WAH compressed bitmap index, the last word of each bitmap, referred to as its active word, is handled separately. It is represented with a literal word and a mask to differentiate bits that are part of the original bitmap from additional padding bits, added to maintain the word alignment.

On a uniformly distributed and very sparse bitmap, $d << 1/(w-1)$, the chances to have only unset bits in the last group are high: $(1-d)^{w-1} \approx 1-(w-1)d \approx 1$. Thus, the total size of the WAH bitmaps in the index is: $2L + 2c$. As long as $c << N = L$, the approximation in Equation 5.13 remains valid. However, for very high cardinality attributes, where the number of elements is lower than the cardinality, the compressed bitmap grows linearly to the attribute cardinality.

With PLWAH, the zero padding is performed before compression, the whole bitmap is thus compressed without breaking the word alignment. This implies that the compressed bitmap carries no information on the precise size of its original uncompressed form. However, since all bitmaps in the index have the same size, only a single word is required to identify the total size of the uncompressed bitmap. As the uncompressed bitmap size is known, it no longer required to store tailing zeros in a bitmap.

PLWAH bitmaps do not store the unset tailing bits of the bitmap, e.g., bitmaps will never finish with a zero fill word and an empty position list. Bitmaps can thus have different lengths, where unrepresented bits are considered to be unset. Bitwise operators are implemented accordingly.

Allowing bitmaps to have different size has many advantages. First, the attribute values having no corresponding elements are not stored in the index. Second, the size of a PLWAH index is: $2L + 1$. Thus, for high cardinality attributes, the size of the bitmap is solely dependent on the number of elements, and not the attribute cardinality, as it was the case with WAH. To illustrate this, Figure 5.18 shows the organization of the bitmap index where the cardinality is higher than the number of elements. The PLWAH index choses only to represent bitmaps for existing values. In a classic bitmap index, or with WAH, all the values are represented.

Third, to add an element, only the bitmap corresponding to the value of the indexed attribute needs to be modified; a set bit must is appended to the uncompressed bitmap. In a PLWAH bitmap, this is done by generating a bitmap corresponding to the element, i.e., a bitmap composed of a single zero fill word and a position list of size 1, and executing an OR operation between the new bitmap and the existing bitmap. The counter of the total number of element indexed is increased by one, we can thus keep track of the size of the uncompressed bitmaps. The complexity of in-

key	att1	att2
1	18	0
2	88	-6,000
3	18	0
4	88	18
5	88	88
...		

att#	value	PLWAH bitmap
1	18	[1,3]
1	88	[2,4,5]
2	-6,000	[2]
2	0	[1,3]
2	18	[4]
2	88	[5]
...		

Figure 5.18: Example of a PLWAH Bitmap Index

serting an element to a PLWAH bitmap index is thus constant, where with WAH, all bitmaps need to be updated, i.e., the time complexity is proportional to the attribute cardinality.

5.B Memory Allocation

Memory allocation is a major bottleneck for performing fast operations on long range queries. Long range queries are best performed using in-place bitwise operators between a uncompressed and a compressed bitmap. An uncompressed bitmap requires a lot of memory, it is thus important to reserve all the memory necessary at once, although avoiding to reserve too much memory. Since we keep track of the total of elements in the index, the precise size of the uncompressed bitmap can easily be calculated.

For operations on shorter range queries, memory is allocated for each operation. This can be avoided by allocating memory for two bitmaps and recycling the memory used as input by the previous operation to store the resulting bitmap of the current operation. This is future work.

Algorithm 5.6 Recycling Memory for Bitwise Operation on Compressed Bitmaps

RecOR(integer a, integer b, List of compressed bitmap B)
1: $previous \leftarrow$ memory allocation ▷ Initial memory allocation
2: $current \leftarrow$ memory allocation ▷ Initial memory allocation
3: **for all** i in $[a, b]$ **do**
4: swap($current$,$previous$);
5: $current$ = OR(B_i, $previous$);
6: **return** $current$

For operation on compressed bitmaps, an upper bound on the memory required can be calculated for each operation. However, in order to speed-up the operations,

the memory should be allocated at once for the complete range, i.e., for the series of OR operations corresponding to a complete coverage of the range. However, no upper bound can be calculated without sequentially collecting the size of bitmaps. We implemented a dedicated bitmap memory allocation library and experimented different heuristics in order to minimize the memory size allocated and minimize the number of allocation required.

5.C Reading with One Branch

A key aspect of PLWAH is that it improves significantly the compression ratio without adding a performance penalty. This is achieved by having a very efficient reading operation, which requires the same amount of branches that the regular WAH compression. The decompression of a word is performed with only one branching operation. The modification of a position list to a sparse literal word is done only with a series of bitshift operations. The details of these operation are shown in Algorithm 5.7.

In fact, compared to the original WAH implementation, the number of branches has been reduced. However, in order to keep the comparison fair, the branching optimization were back-ported to the original WAH implementation for all the experiments.

5.D Various Bitwise Operators

The various OR, AND, XOR, NOTAND operators were implemented for the bitmaps. Furthermore, a 'generic' function that can accommodate any bitwise operator is coded. However, the generic bitwise function is not as efficient as dedicated operators for 2 reasons: (1) extra computation is necessary to cover all possible cases; and (2) the bitwise operator requires a pointer to a function, thus making "inlining" impossible and causing an significant overhead since the bitwise function call is repeated once or twice per iteration in the main loop.

5.E Bulk Load Insertion

Using the in-place operator, each of the c bitmap being built is kept uncompressed in memory. For large data set or high cardinality attribute, this approach allocates all memory available and requires memory swaps that result in serious performance drops.

A second approach to construct the index is to read sequentially each attribute, to generate a bitmap with its key, and to OR the newly generated bitmap with the exist-

Algorithm 5.7 Decompressing with a Limited Number of Branches

Decompress(current run run, word $word$)
1: **if** $word$ is counter **then**
▷ "<<" denotes a logical left bitshift
▷ ">>>" denotes an arithmetic right bitshift
2: $run.data \leftarrow (word << 1) >>> 63$
3: $run.nWords \leftarrow word\&MASK\_COUNTER$
4: $run.isFill \leftarrow 1$
5: $run.isSparse \leftarrow (word\&MASK\_POSITION)! = 0$
6: $run.sparse \leftarrow$Positions($word$)
7: **else**
8: $run.data \leftarrow word\&MASK\_LITERAL$
9: $run.nWords \leftarrow 1$
10: $run.isFill \leftarrow 0$
11: $run.isSparse \leftarrow 0$

Position1(word i)
1: **return** $(i >> 56)\&0x003F$

Position2(word i)
1: **return** $(i >> 50)\&0x003F$

Position3(word i)
1: **return** $(i >> 44)\&0x003F$

Position4(word i)
1: **return** $(i >> 38)\&0x003F$

Position5(word i)
1: **return** $(i >> 32)\&0x003F$

Positions(word i)
1: $tmp \leftarrow (0x8000000000000001ULL << Position1(i))$
2: $|(0x8000000000000001ULL << Position2(i))$
3: $|(0x8000000000000001ULL << Position3(i))$
4: $|(0x8000000000000001ULL << Position4(i))$
5: $|(0x8000000000000001ULL << Position5(i))$
6: **return** $tmp >> 1$

ing bitmap corresponding to the value. However, this process results in unnecessary compression and decompression.

A third and preferred approach for the initial construction of a PLWAH index on a large dataset, is to first construct an array of all keys corresponding to each value, and then to transform the array and to compress it at once. The array is generated using

a custom memory pre-allocation scheme, that avoids allocation to be performed each time a new element is inserted.

For very large data set, the second method is, however, preferred as the memory allocation keeps in memory only 2 bitmaps at a time. The third method is, by far, the most efficient, if all the compressed index can be stored in memory.

5.F Adaptive Counter Size

A potentially significant problem with the PLWAH compression technique for 32 bit words occur when the number of elements significantly exceeds 10^9, which is not uncommon for some types of applications. In this case, the very long runs occuring for high cardinality attributes cannot be represented with a single fill word. Instead, for a run with a length of 10^{11}, a chain of approximately 100 fill words is needed. This might be resolved by using 64 bit words, but this immediately doubles the size of each word in the index (and thus more or less doubles the total index size, too), in order to solve a problem that might not occur for most runs. Instead, we propose a technique called *adaptive counter size* that basically uses 32 bit words, when this is enough, and then adapts to 64 bit words *when necessary*, meaning that the longer words are only used for very long runs.

This can be achieved without changing the basic encoding scheme. A very long run will be encoded with an empty position list in the (first) 32 bit (fill) word. The next word will then also become a fill word, and the total length of the run (using a 50 bit counter) will be encoded in two parts: the first 25 LSBs of the length will be put into the counter part of the first word, while the other 25 MSBs of the length will be put into the counter part of the second word. When decoding, Algorithm 5.8 reads the first word and sets the variable *isSparse* (along with other variables) as usual. However, the value of the word type and the isSparse variable are used by the following word to detect the presence of an adaptive counter.

This works as follows. We know (due to the branch in Line 1) that the present word is a fill word. We decode the fill word type as a temporary value. There are now two mutually exclusive cases:
1) The current word is part of an adaptive counter, that is, the previous word was a fill word with an empty position list and was of the same type than the current word.
2) The currentword is part not part of an adaptive counter, that is, the previous word was a literal word, or was a fill word of a different type.
In case 1) the counter is updated, considering that it represents the 25 MSB of a 50 bit long counter. In case 2) the counter is updated, considering that it represents the 25 LSB of a 50 bit long counter and the word fill type is update with the temporary value. The $nWords$ variable is then in effect updated with the value of the 50 bit counter, but over *two* calls of Algorithm 5.8. The first call returns the LSBs part, while the 2nd

Algorithm 5.8 Reads a compressed word with adaptive counter

```
    ReadWord ( Compressed word W )
 1: if W is a fill word then
 2:     tmpdata ← ((W >> 30)&1) * all_ones
 3:     nWords ← (W&counter_mask) << (25 * (tmpdata = data) * (isSparse = false))
 4:     data ← tmpdata
 5:     isFill ← true
 6:     isSparse ← (W&position_mask)
 7:     sparse ← bitmap constructed from the position list
 8: else
 9:     data ← W&first_bit_unset                          ▷ MSB of W is unset
10:     nWords ← 1
11:     isFill ← false
12:     isSparse ← false
```

call returns the MSBs part. Thus, no change is required in the code calling 5.8. For this to work properly even for the first word, $isSparse$ must be initialized to $false$ and $data$, with a value different to the fills before the first call of Algorithm 5.8.

An alternative way of implementing adaptive counter size is to use one or more extra bits in the "header" of the word (after the word type bit and the fill type bit) to represent how long the counter is. This can also be used to determine whether the position list part of the word is used for position lists or, instead, used to increase the number of bits available for the counter. This requires a corresponding update of the decoding logic.

Chapter 6

Using Fuzzy Lists for Playlist Management

The increasing popularity of music recommendation systems and the recent growth of online music communities further emphasizes the need for effective playlist management tools able to create, share, and personalize playlists. This chapter proposes the development of generic playlists and presents a concrete scenario to illustrate their possibilities. Additionally, to enable the development of playlist management tools, a formal foundation is provided. Therefore, the concept of fuzzy lists is defined and a corresponding algebra is developed. Fuzzy lists offer a solution perfectly suited to meet the demands of playlist management.

6.1 Introduction

The proliferation of broadband Internet connections and the development of new digital music formats have led to the explosion of online music communities and music recommendation systems. The increasing popularity of these systems has created a strong demand for the development of *playlist manipulation engines*. Hence, playlist models are highly needed. However, playlists are by nature *imprecise*. One song could probably be replaced by another, while preserving the essence of the playlist. Similarly, two songs can sometimes be permuted. The way playlists are built explains this phenomenon. For individual music lovers, the manual construction of a playlist results in some kind of consensus between the various aspects defining the songs [29, 66]. In large automated music recommendation systems, user collaborative filtering and co-occurrence analysis approaches are commonly used to construct

playlists [27]. While imprecise, each playlist can exactly characterize a trend, a dynamic, a mood. Additionally, playlists also have a *subjective* nature, i.e., they are highly dependent on their audience. Listeners might have strong musical preferences or may not have access to all the music. Therefore, playlist management tools have to include *personalization mechanisms*.

The contributions of this paper are twofold. (i) A new scheme for constructing and sharing playlists is described via a concrete scenario. The scenario illustrates how the imprecise and subjective characteristics of playlists can be handled in order to improve playlist management engines. (ii) *Fuzzy lists*, a generalization of lists and fuzzy sets, are defined and their corresponding algebra is developed. The proposed algebra is inspired by relational algebra in order to facilitate future implementation in an RDBMS. Fuzzy lists offer the formal foundation for the development of playlist management tools as the provided examples illustrate.

The remainder of this paper is organized as follows. The creation of fuzzy lists for playlist management is motivated by a scenario presented in Section 6.2. Section 6.3 discusses the related work. Section 6.4 provides a formal definition of fuzzy lists, their operators and functions. Concrete playlist manipulation examples are shown to underline their utility. Finally, after evoking some implementation considerations in Section 6.5, the conclusions and suggestions for future work are presented in Section 6.6.

6.2 Motivation

In this section, a playlist management engine that respects both the imprecise and subjective nature of playlists is envisioned. The users of the system want to create and share playlists. The playlists are built by user communities in a collaborative fashion rather than by individual users. Automated classification systems could possibly be incorporated, as well as classical users participating in the playlist building process. Additionally, the generated playlists are adaptable to the users' profiles to respect their musical taste. The setup is as follows.

The objective is to create a playlist composed of a given number of songs and a theme. For example, 100 users are asked to build a playlist composed of ten songs, the first 3 songs should be rock, the next 3 should sound jazzy, the following 3 romantic, and the last one should be a blues song. Furthermore, the users are asked to provide smooth transitions between the songs, e.g., the third rock song should sound a bit jazzy. Agreement between the different users is achieved thanks to a voting mechanism. Finally, independently from the playlist building process, registered users have lists of songs they like and dislike.

The playlists created by the voters are merged into a single playlist, referred to as a *generic playlist*. The generic playlist stores the "election score," referred to

as the *membership degree*, obtained for each song and each position. The generic playlist can then be shared among all the registered users of the system. However, the generic playlist cannot be used directly by the users, as for each position in the playlist, many songs will probably coexist. Furthermore, if some songs have been previously tagged as the user's favorites, then they should preferably be played, or if the user has no access to a song, an accessible song should be alternatively chosen. The personalization mechanism selects songs from the generic playlist with respect to the user preferences and constraints. For each possible song, a preference grade is given; songs with a high grade should preferably be played while songs with a low grade should be played less often. A selection score is calculated using the user's preference list and the generic playlist. The songs with the highest score win. Figure 6.1 illustrates the overall construction of a generic playlist and how it is derived into a personalized playlist a latter stage.

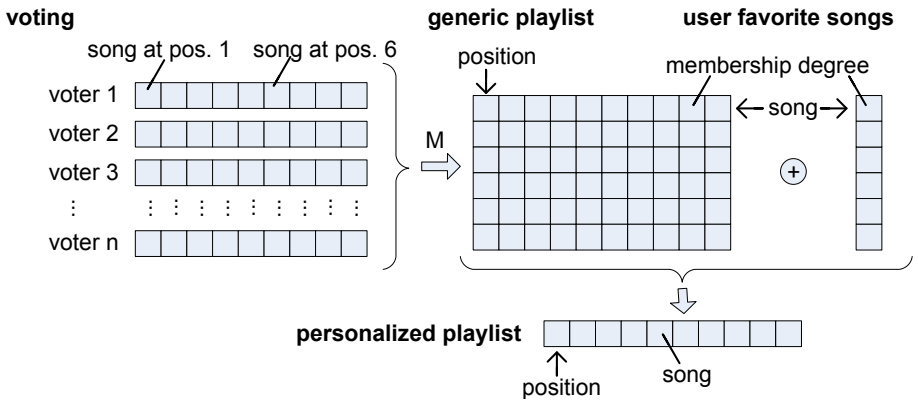


Figure 6.1: Generic Playlists Usage Scenario

The merging and personalization mechanisms will now be described and illustrated by a concrete example. The functions presented were chosen for their simplicity, as the aim here is to motivate the use of generic playlists.

Table 6.1 represents the vote counts for the creation of a playlist with a length of 10 songs where the base song set is composed of 12 songs. The merging function, denoted M, grants a low membership degree to songs that have receive only a few votes, and a high membership degree to songs that have received many votes, at a given position. Let the generic playlist A, shown in Table 6.2, be generated by applying M to Table 6.1.

Assume a user u has rated his music preferences using a fuzzy song set as follows. The user's preference list, denoted F_u, assigns to each song in the system a preference

song	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
s1	50	20	20	10	0	0	0	0	0	0
s2	20	50	30	0	0	0	0	0	0	0
s3	20	20	40	0	0	0	0	0	0	0
s4	10	10	10	10	0	0	0	0	0	0
s5	0	0	0	20	20	0	0	0	0	0
s6	0	0	0	60	40	0	0	0	0	0
s7	0	0	0	0	40	20	30	30	0	0
s8	0	0	0	0	0	80	20	0	0	0
s9	0	0	0	0	0	0	40	60	10	0
s10	0	0	0	0	0	0	10	10	10	70
s11	0	0	0	0	0	0	0	0	80	30
s12	0	0	0	0	0	0	0	0	0	0

Table 6.1: Users votes count

song	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
s1	0.5	0.2	0.2	0.1	0	0	0	0	0	0
s2	0.2	0.5	0.3	0	0	0	0	0	0	0
s3	0.2	0.2	0.4	0	0	0	0	0	0	0
s4	0.1	0.1	0.1	0.1	0	0	0	0	0	0
s5	0	0	0	0.2	0.2	0	0	0	0	0
s6	0	0	0	0.6	0.4	0	0	0	0	0
s7	0	0	0	0	0.4	0.2	0.3	0.3	0	0
s8	0	0	0	0	0	0.8	0.2	0	0	0
s9	0	0	0	0	0	0	0.4	0.6	0.1	0
s10	0	0	0	0	0	0	0.1	0.1	0.1	0.7
s11	0	0	0	0	0	0	0	0	0.8	0.3
s12	0	0	0	0	0	0	0	0	0	0

Table 6.2: Generic Playlist

song	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
s1	0.50	0.35	0.35	0.30	0.25	0.25	0.25	0.25	0.25	0.25
s2	0.30	0.45	0.35	0.20	0.20	0.20	0.20	0.20	0.20	0.20
s3	0.30	0.30	0.40	0.20	0.20	0.20	0.20	0.20	0.20	0.20
s4	0.05	0.05	0.05	0.05	0.00	0.00	0.00	0.00	0.00	0.00
s5	0.20	0.20	0.20	0.30	0.30	0.20	0.20	0.20	0.20	0.20
s6	0.35	0.35	0.35	0.65	0.55	0.35	0.35	0.35	0.35	0.35
s7	0.30	0.30	0.30	0.30	0.50	0.40	0.45	0.45	0.30	0.30
s8	0.25	0.25	0.25	0.25	0.25	0.65	0.35	0.25	0.25	0.25
s9	0.25	0.25	0.25	0.25	0.25	0.25	0.45	0.55	0.30	0.25
s10	0.25	0.25	0.25	0.25	0.25	0.25	0.30	0.30	0.30	0.60
s11	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.65	0.40
s12	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35

Table 6.3: Modified generic playlist

score reflecting if the user likes, is neutral to, or dislikes the song.

$$F_u = \{0.5/s1, 0.4/s2, 0.4/s3, 0.0/s4, 0.4/s5, 0.7/s6,$$
$$0.6/s7, 0.5/s8, 0.5/s9, 0.5/s10, 0.5/s11, 0.7/s12\}$$

The generalized playlist and the user's preferences are used to construct the modified generic playlist, presented in Table 6.3. For each song at a given position in the generic playlist, the average between the membership degree of the generic playlist and the user's preference score is calculated.

Finally, the personalized playlist, denoted P, is generated by selecting, for each position, the song with the highest membership degree.

$$P = [s1, s2, s3, s6, s6, s8, s7, s9, s11, s10]$$

The personalized playlist generated is sensible: $s12$ that has not received any votes is not present; $s4$ that has a preference score of zero, e.g., the user does not own the song, is not present; $s6$ and $s8$ that have the highest preference score are present; and the rest of the songs are at placed in accordance with the votes. The consecutive repetition of song $s6$ could be avoided by using a slightly more sophisticated personalization method, e.g., by lowering the preference score of songs previously selected at a nearby position.

The merging and personalization functions used in the previous example were solely chosen for their simplicity. Experiments on a real system should be conducted to obtain reasonable weight estimates for both the creation of generic playlists from multiple sources and their personalization. However, the correct weights to be used, if they can be estimated, are domain specific and are considered as external parameters.

Proportional positioning of the song elements, where positions are specified with respect to the whole playlist, e.g., a song is located at 20% of the playlist length, is commonly found in playlist management systems. Proportional positioning offers relative comparisons that a simple index of the songs positions does not capture, e.g., blues songs are generally located in the second third of the playlists of given group of users. Fuzzy lists can simply be adapted to support proportional positioning by transforming the generic playlist to generic playlists having a common length, e.g., by dividing the position of each element by the total length m and multiplying it by a reference length, say, 100. The algebra proposed in Section 6.4 supports transformations that accommodate both playlists shorter and longer than the reference length. More generally, relative positioning of elements, where elements are positioned relatively to each other, can be captured by functions such as, for example, sliding windows that take into account membership degrees of songs in a given neighborhood position in a generic playlist.

If generic playlists were modeled with *ordinary lists*, then the membership degrees would either be 0 or 1, and all songs that have received votes would have an identical probability of being part of the playlist. Instead, if *fuzzy sets* were used to model generic playlists, then the position dimension of the playlist would be lost and all songs would have the same probability of being part of the playlist, regardless of the position. Thus, *fuzzy lists* are the best choice to model playlists as they capture the interdependency between song, position, and membership degree.

6.3 Related Work

Recently, work on fuzzy sets was reported for modeling music similarities and user preferences [30]. Three different storage schemes are discussed. This paper re-uses the idea of user favorite song sets to provide playlist personalization features. Fuzzy lists, a generalization of lists and fuzzy sets, were studied previously [88]. However, the approach described is not appropriate to represent playlists for two reasons. First, some positions may be undefined, i.e., they have no corresponding elements or membership degrees. Second, multiple membership degrees may be defined for a single element at a given position. By contrast, the present paper allows an element to have the same membership degree at different positions, e.g., songs having a zero membership degree regardless of their position in the generic playlist.

Previous work on playlist generation dealt with algorithms to efficiently find a playlist which fulfills given constraints [12, 3, 74]. Work on dynamic playlist generation where songs are retrieved one at a time and listeners can intervene in the playlist creation is presented by Pampalk et al. [69]. A data and query model for dynamic playlist generation was proposed by Jensen et al. [49]. The authors are able to retrieve songs similar to a given seed while avoiding songs with respect to the user preferences. However, no solution is proposed to the issues of sharing and personalizing existing playlists. Additionally, the playlists are created in a song by song fashion and aggregating playlists to capture mood or genre similarities is not possible. However, some applications require functionalities, e.g., searching missing songs in a playlist, that require capturing the essence of the playlist [45].

A traditional approach to store musical information in a database is to use classical relational models such as the one proposed by Rubenstein [79]. The model extends the entity-relationship data model to implement the notion of hierarchical ordering, commonly found in musical data. Wang et al. have presented a music data model, its algebra, and query language [90]. The data model is able to structure both the musical content and the metadata. However, all these models are limited to individual music performances and do not cover playlists.

Finally, the most related work is probably the foundation for query optimization on ordered sets proposed by Slivinskas et al., where a list-based relational algebra is presented [83]. However, the framework does not cover either multiple elements coexisting at a given position of the list, or membership degrees. Furthermore, the framework does not address playlist management issues.

6.4 The Fuzzy List Algebra

This section provides a formal foundation for generic playlists. A definition of fuzzy lists is first provided, followed by a description of fuzzy list operators. The opera-

tors are divided in three categories. Operators similar to the list operators are first presented, followed by unary operators, and finally by binary operators.

6.4.1 Definition

A finite fuzzy list, A, of length m_A over a domain X is defined as follows.

$$A = \{\mu_A(x,n)/n/x \mid x \in X, n \in \{1,\ldots,m_A\}, \mu_A : X \times \mathbb{N} \mapsto [0;1]\}$$

Here x is an element of X, n is a non-negative integer, and $\mu_A(x,n)$, referred to as the sequential membership degree of x at position n, is a real number belonging to $[0;1]$. $\mu_A(x,n) = 0$ when case x does not belong to A at position n, and $\mu_A(x,n) = 1$ when x completely belongs to A at position n. The length of A, length(A), is defined as m_A.

6.4.2 List-like Operators

In the following, let A, A_1, and A_2 be three fuzzy lists defined over a domain X.

6.4.2.1 Equality

A_1 and A_2 are equal iff A_1 and A_2 have the same length and have the same membership degree between identical pairs of element and position:

$$\begin{aligned} A_1 = A_2 \Leftrightarrow \text{length}(A_1) = \text{length}(A_2), \text{ and} \\ \forall x \in X, n \in \{1,...,\text{length}(A_1)\} : \mu_{A_1}(x,n) = \mu_{A_2}(x,n) \end{aligned}$$

6.4.2.2 Sublist

A_1 is a fuzzy sublist of A_2 iff a sequence exists in A_2 where all the membership degrees of A_1 are less than or equal to the membership degrees of A_2:

$$\begin{aligned} A_1 \subseteq A_2 \Leftrightarrow \exists i \in \mathbb{N} \mid \forall x \in X, \forall n \in \{1,...,\text{length}(A_1)\} : \\ \mu_{A_1}(x, n+i) \leq \mu_{A_2}(x,n) \end{aligned}$$

Note that the empty fuzzy list of length 1, denoted ϕ_1, is a fuzzy sublist of all fuzzy lists. Note also that $A_1 = A_2 \Leftrightarrow A_1 \subseteq A_2$ and $A_2 \subseteq A_1$. In the playlist context, a sublist is a sequence of a playlist where the probability of a song being played at a given position is lowered. Songs not present in a generic playlist will remain excluded in all its sublists.

6.4.2.3 Concatenation

The concatenation of A_1 and A_2 is the fuzzy list of length(A_1) + length(A_2) where the fuzzy list A_2 succeeds A_1 as follows.

$$A_1 \| A_2 = \{\mu_{\|}(x,n)/n/x \mid \forall x \in X, \forall n \in \{1,...,\text{length}(A_1)+\text{length}(A_2))\} : \\ \mu_{\|}(x,n) = \begin{cases} \mu_{A_1}(x,n) & \text{for } n \leq \text{length}(A_1), \\ \mu_{A_2}(x, n-\text{length}(A_1)) & \text{for } n > \text{length}(A_1) \end{cases} \}$$

Note that $A_1 \subseteq (A_1 \| A_2)$ and $A_2 \subseteq (A_1 \| A_2)$. The concatenation operator is inspired by the UNION ALL operator in SQL. In the playlist context, the concatenation operator allows short playlists to be used as the building blocks of longer playlists, e.g., to propose playlists constructed based on the succession of different generic playlists.

6.4.3 Unary Operators

The operators presented below modify the position, e.g., to reorder a generic playlist, and the nature, e.g., to capture the likelihood of an artist to be played rather than a song, of the elements.

6.4.3.1 Unary Reordering, Selection, and Aggregation

The unary aggregation is a generalization of the unary selection that is, in turn, a generalization of the unary reordering. For ease of understanding, the three operators are progressively introduced. They transform the fuzzy list based on the positions of the elements. Each of the operators is defined by a different mapping function as illustrated in Figure 6.2.

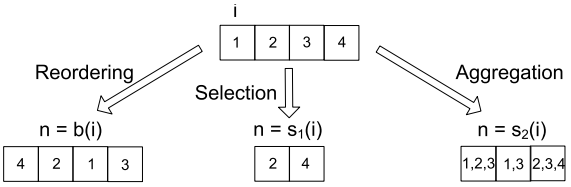


Figure 6.2: Examples of Position Mapping for Unary Reordering, Selection and Aggregation Operators

Unary Reordering:
Given a bijection $b : \{1,\ldots,\text{length}(A)\} \leftrightarrow \{1,\ldots,\text{length}(A)\}$, the bijection of A

with respect to b is defined as follows.

$$A_b = \{\mu_{A_b}(x,n)/n/x \mid \forall x \in X, \forall n \in \{1,\ldots,\text{length}(A)\} : \\ \mu_{A_b}(x,n) = \mu_A(x, b^{-1}(n))\}$$

A unary reordering of a fuzzy list is defined by a permutation, and allows the creation of, e.g., a shuffling function that randomly reorders the playlist. The *invert* operator defined as follows, is another example of unary ordering.

$$\text{Invert}(A) = \{\mu_{\text{Invert}}(x,n)/x/n \mid \forall x \in X, \forall n \in \{1,\ldots,\text{length}(A)\} : \\ \mu_{\text{Invert}}(x,n) = \mu_A(x, \text{length}(A) - n)\}$$

Unary Selection:
Given a set $A' \subseteq \{1,\ldots,\text{length}(A)\}$, and a bijection $s_1 : A' \leftrightarrow \{1,\ldots,\text{size}(A')\}$, a selection of A over s_1 is defined as follows.

$$A_{s_1} = \{\mu_{A_{s_1}}(x,n)/n/x \mid \forall x \in X, \forall n \in \{1,\ldots,\text{size}(A')\} : \\ \mu_{A_{s_1}}(x,n) = \mu_A(x, s_1^{-1}(n))\}$$

A reordering is a particular case of a selection where $A' = \{1,\ldots,\text{length}(A)\}$. The selection operator removes the elements at a given position of the fuzzy list. Therefore, if s_1 is chosen to verify a given predicate, only the elements that fulfill the predicate will be kept. Thus in a playlist, the positions in the generic playlist where the songs made by U2 have a high membership degree could be removed or kept.

Unary Aggregation:
Given a set $A' \subseteq \{1,\ldots,\text{length}(A)\}$, and a surjective mapping $s_2 : A' \mapsto \{1,\ldots,\max(s_2)\}$, let S_n be the set of all fuzzy list elements $\{\mu(x,i)/i/x\}$ of A with $n = s_2(i)$, and let $a : 2^{S_n} \mapsto [0;1]$ be a surjective mapping. The aggregation of A with a over s_2 is defined as follows.

$$A^a_{s_2} = \{\mu_{A^a_{s_2}}(x,n)/n/x \mid \forall x \in X, \forall n \in \{1,\ldots,\max(s_2)\} : \\ \mu_{A^a_{s_2}}(x,n) = a(S_n)\}$$

A selection is a particular case of an aggregation where $A' = \{1,\ldots,\text{length}(A_1)\}$ and $a(S_n) = \mu(x, s_2^{-1}(n))$. Aggregations can be used to reinforce the membership degree of certain songs at a given position in a generic playlist if similar songs also have a high membership degree in nearby positions. Aggregations can also be used to generate an overview over different positions in a generic playlist, e.g., an average over a sliding window. The unary aggregation operator is similar to aggregations in SQL.

6.4.3.2 Projection

Let Y be a set of elements, and let y be one of its elements. Let $p : X \mapsto Y$ be a surjection, let S_y be the set of all fuzzy list elements $\{\mu(x,n)/n/x\}$ of A with $x = p^{-1}(y)$, and let $a : 2^{S_y} \mapsto [0;1]$ be a function. The projection of A with respect to p and a is defined as follows.

$$\Pi_p^a(A) = \{\mu_\Pi(y,n)/n/y \mid \forall n \in \{1,\ldots,\text{length}(A)\}, \forall y \in Y : \\ \mu_\Pi(y,n) = a(S_y)\}$$

Since p is a surjection, $\mu_\Pi(y,n)$ is defined $\forall y \in Y$. The projection operator allows grouping elements of the fuzzy lists, e.g., it is possible to obtain an overview of the generic playlist in terms of artists or genre categories by mapping each song to at least one or more artists or genres. The mapping is specified by p and the new membership degree is determined by a.

6.4.4 Binary Operators

The following operators are defined over two fuzzy lists. They allow generic playlists to be merged, aggregated, or compared.

6.4.4.1 Binary Reordering, Selection and Aggregation

Binary operators allow merging two fuzzy lists, e.g., two playlists, into one by specifying an ordering, i.e., the position of the songs, and optionally how the membership degrees should be changed, i.e., the likelihood for a song to be selected in the playlist. To define binary operators, three position mapping functions b, s_1, and s_2 are used as illustrated in Figure 6.3. As presented earlier in the case of unary operators, the binary reordering, selection and aggregation will be successfully introduced for clarity reasons. Binary aggregations are a generalization of binary selections that are, in turn, a generalization of binary reorderings.

Binary Reordering:
Let $b : \{1,\ldots,\text{length}(A_1)\} \times \{1,\ldots,\text{length}(A_2)\} \leftrightarrow \{1,\ldots,\max(b)\}$ be a bijection as, e.g., illustrated by Figure 6.3. Let $\star : [0;1] \times [0;1] \mapsto [0;1]$ be a binary operator. The binary reordering between A_1 and A_2 with b and $\star$ is defined as follows.

$$A_1 \underset{b}{\star} A_2 = \{\mu_\star(x,n)/n/x \mid \forall x \in X_1, \forall n \in \{1,\ldots,\max(b)\}, \\ \forall (i,j) = b^{-1}(n) : \mu_\star(x,n) = \mu_{A_1}(x,i) \star \mu_{A_2}(x,j)\}$$

Common operators are for example the product, the minimum, the maximum, and the average. The average, e.g., of two generic playlists built by different user groups

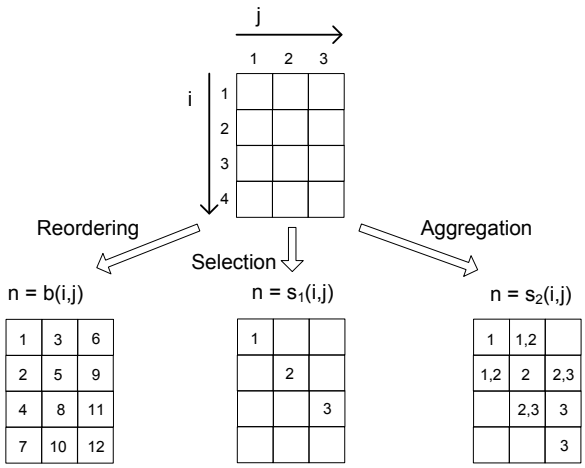


Figure 6.3: Examples of Position Mapping for Binary Reordering, Selection and Aggregation operators

that were given the same constraints, can be performed as follows.

$$A_1 \underset{b}{\text{AVG}} A_2 = \{\mu_{\text{AVG}}(x,n)/n/x \mid \forall x \in X_1, \forall n \in \{1,\ldots,\max(b)\},$$
$$\forall (i,j) = b^{-1}(n) : \mu_{\text{AVG}}(x,n) = \frac{\mu_{A_1}(x,i) + \mu_{A_2}(x,j)}{2}\}$$

Binary Selection:
Let $A_1' \subseteq \{1,\ldots,\text{length}(A_1)\}$ and $A_2' \subseteq \{1,\ldots,\text{length}(A_2)\}$ be two sets. Let $s_1 : \{1,\ldots,\text{size}(A_1')\} \times \{1,\ldots,\text{length}(A_2')\} \mapsto \{1,\ldots,\max(s_1)\}$ be a bijection as, e.g., illustrated in Figure 6.3, and let $\star : [0;1] \times [0;1] \mapsto [0;1]$ be a binary operator. The binary selection of A_1 and A_2 with s_1 and $\star$ is defined as follows.

$$A_1 \underset{s_1}{\star} A_2 = \{\mu_\star(x,n)/n/x \mid \forall x \in X, \forall n \in \{1,\ldots,\max(s_1)\},$$
$$\forall (i,j) = s_1^{-1}(n) : \mu_\star(x,n) = \mu_{A_1}(x,i) \star \mu_{A_2}(x,j)\}$$

Binary selections over fuzzy lists are very common operations. Intersection and union operators are particular cases where the $\star$ operators are respectively the minimum and the maximum, with some corresponding changes to the position mapping function s_1.

The *intersection* of A_1 and A_2 is defined as a fuzzy list, where the membership degree for any element at a particular position is its minimum value in A_1 and A_2.

$$A_1 \cap A_2 = \{\mu_\cap(x,n)/n/x \mid \forall x \in X, \forall n \in \{1,...,\min(\text{length}(A_1),\text{length}(A_2))\} :$$
$$\mu_\cap(x,n) = \min(\mu_{A_1}(x,n), \mu_{A_2}(x,n))\}$$

Note the following properties: $(A_1 \cap A_2) \subseteq A_1$, $(A_1 \cap A_2) = (A_2 \cap A_1)$, $A_1 \cap \phi_1 = \phi_1$, and $A_1 \cap A_1 = A_1$.

The *union* of A_1 and A_2 is defined as a fuzzy list, where the membership degree for any element at a particular position is set to its maximum value in A_1 and A_2.

$$A_1 \cup A_2 = \{\mu_{\cup}(x,n)/n/x \mid \forall x \in X, \forall n \in \{1, ..., \max(\text{length}(A_1), \text{length}(A_2))\} :$$
$$\mu_{\cup}(x,n) = \begin{cases} \mu_{A_1}(x,n) & \text{for } n > \text{length}(A_2) \\ \mu_{A_2}(x,n) & \text{for } n > \text{length}(A_1) \\ \max(\mu_{A_1}(x,n), \mu_{A_2}(x,n)) & \text{otherwise} \end{cases} \}$$

Note the following properties: $A_1 \subseteq (A_1 \cup A_2)$, $(A_1 \cup A_2) = (A_2 \cup A_1)$, $A_1 \cup \phi_1 = A_1$, and $A_1 \cup A_1 = A_1$.

Intuitively, the intersection operator returns a generic playlist where only the songs strongly present in both of the two provided generic playlists will be very present in the resulting one. The intersection and the union operators are commonly used to build new generic playlists from existing ones.

Binary Aggregation:
Let $A'_1 \subseteq \{1, \ldots, \text{length}(A_1)\}$ and $A'_2 \subseteq \{1, \ldots, \text{length}(A_2)\}$ be two sets, let $s_2 : \{1, \ldots, \text{length}(A'_1)\} \times \{1, \ldots, \text{length}(A'_2)\} \mapsto \{1, \ldots, \max(s_2)\}$ be a surjective mapping as, e.g., illustrated in Figure 6.3, and let $\star : [0;1] \times [0;1] \mapsto [0;1]$ be a binary operator. Let S_n be the set of all the pairs of fuzzy lists elements $(\mu_{A_1}(x_{A_1}, i); i; x_{A_1})$ and $(\mu_{A_2}(x_{A_2}, j); j; x_{A_2})$ so that $s_2(i,j) = n$. Let $a(S_n) : 2^{S_n} \mapsto [0;1]$ be a function. The aggregation of A_1 and A_2 with a and s_2 is defined as follows.

$$A_1 \underset{s_2}{\overset{a}{}} A_2 = \{\mu^a_{s_2}(x,n)/n/x \mid \forall x \in X, \forall n \in \{1, \ldots, \max(s_2)\} :$$
$$\mu^a_{s_2}(x,n) = a(S_n)\}$$

For example, the average of two generic playlists can be performed over two sliding windows as illustrated by $s2$ in Figure 6.3. Such an average, less sensitive to small position differences between the two generic playlists, can be defined as follows.

$$s_2(i,j) = n \mid i = j, \{(i-1,j), (i,j-1), (i,j), (i,j+1), (i+1,j)\} \mapsto n$$
$$a(S_n) = \frac{\sum_{(i,j)=s_2^{-1}(n)} \mu_{A_1}(x,i) + \mu_{A_2}(x,j)}{\sum_{(i,j)=s_2^{-1}(n)} 1}$$

The binary aggregation operator is an adaptation to fuzzy lists of the join and aggregation operators found in SQL.

6.4.4.2 Cartesian Product

Let A_1 and A_2 be two fuzzy lists defined over the two domains X_1 and X_2. Let $s : \{1, \ldots, \text{length}(A_1)\} \times \{1, \ldots, \text{length}(A_2)\} \mapsto \{1, \ldots, \max(s)\}$ be a surjective

mapping and let $S_n : \{((\mu_{A_1}; i; x_{A_1}); (\mu_{A_2}; j; x_{A_2}))\}$ be the set of all the pairs of fuzzy lists elements so that $s(i, j) = n$. The Cartesian product of A_1 and A_2 with respect to a and s is defined as follows.

$$A_1 \overset{a}{\underset{s}{\times}} A_2 = \{\mu_{\times}(x_{A_1 \times A_2}, n)/n/x_{A_1 \times A_2} \mid \forall x_{A_1 \times A_2} \in X_1 \times X_2, \\ \forall n \in \{1, \ldots, \max(s)\} : \mu_{\times \frac{a}{s}}(x_{A_1 \times A_2}, n) = a(S_n)\}$$

Typical selection and aggregation functions for the Cartesian product are, for example, $s(i, j) = i = j$ and $a(S_n) = \mu_{A_1}(x_{A_1}, i) \cdot \mu_{A_2}(x_{A_2}, j)$, that computes the probability of two songs to be played at identical positions in two independent generic playlists.

The Cartesian product operator for fuzzy lists is related to the Cartesian product in SQL. As joins in SQL, binary aggregations are derived operators that can be expressed using projections, selections, and Cartesian products. The concatenation of two fuzzy lists is also a derived operation. Other derived operators inspired from SQL such as Top_k could be useful, e.g., for capturing in a generic playlist the songs that have received the most votes. The fundamental operators of fuzzy lists are: the unary aggregation, the projection, and the Cartesian product.

6.5 Prototyping Considerations

The fuzzy lists and the operators defined above raise many interesting implementation issues. A first implementation option is to develop the fuzzy lists algebra with its own storage, query planner, query executor and to create a database management system, (DBMS), working on fuzzy lists rather than on multi-sets. While this would certainly be a neat solution and provide the greatest efficiency, it would also require to rewrite most parts of the DBMS, from the storage representation, to the query optimizer.

Another approach is to build an abstraction layer that reuses the relation operators defined for multi-sets. Fuzzy lists could be represented as tables, including a position and a membership degree for each tuple. The queries should then be mapped to classical Relational DBMS operators. The main advantage of such an approach is that existing DBMS abstraction and features, e.g., physical storage, query planner and optimizer, can be reused, thus considerably reducing the implementation work. However, efficient storage representations and specific query optimizations will not be available, therefore reducing the scalability of the system. Object Relational DMBSs offer the possibility to specify both storage representations and query optimizations by allowing the definition of Abstract Data Types. However, their underlying algebra still remains confined in a relational model based on multi-sets.

As fuzzy lists are a generalization of lists that are commonly used to represent playlists, the integration of generic playlists into existing workflows of playlist man-

agement systems for playlist representation requires only minor adaptations such as having only 0 or 1 as possible membership degrees. For such usage, using an abstraction layer over a classical Object Relational DBMS is certainly the best approach. Previous work on storage representation for fuzzy song sets [30] using compressed bitmaps is a very interesting starting point to address storage representation issues of generic playlists and fuzzy lists.

6.6 Conclusion and Future work

Generic playlists offer a pragmatic answer to the need for playlist management tools that has recently arisen in online music communities. Generic playlists respect both the consensual and the subjective nature of playlists by defining for each position in the sequence a likelihood degree for the presence of a given song. As illustrated by a concrete scenario, generic playlists are a flexible and concrete solution for constructing, sharing and personalizing playlists. This paper provides a solid foundation for the development of generic playlists by formally defining fuzzy lists and their algebra. Examples of generic playlists motivate the use of the presented fuzzy lists operators. Three basic operators are proposed. Their similarity with relational algebra operators facilitates their implementation in database systems. Future work encompasses research on efficient implementations of the basic operators, exploration of new operators, and experiments on a large scale playlist management engine.

Appendix

In this appendix, we propose an implementation for the storage and common operators of fuzzy lists. The storage implementation adapts the PLWAH compression algorithm in order to compress fuzzy sets rather than representing them using arrays of bitmaps. This representation facilitates the usage of mathematical operators on the membership degrees. The fuzzy list are constructed using the new data storage for fuzzy sets. A pseudo code of the implementation of fuzzy lists major operators and auxiliary data structure is proposed.

6.A Fuzzy List Representation

A fuzzy list can be represented as a list of fuzzy sets defined over the same domain of elements. It can thus be implemented as an array of fuzzy sets as depicted in Figure 6.4.

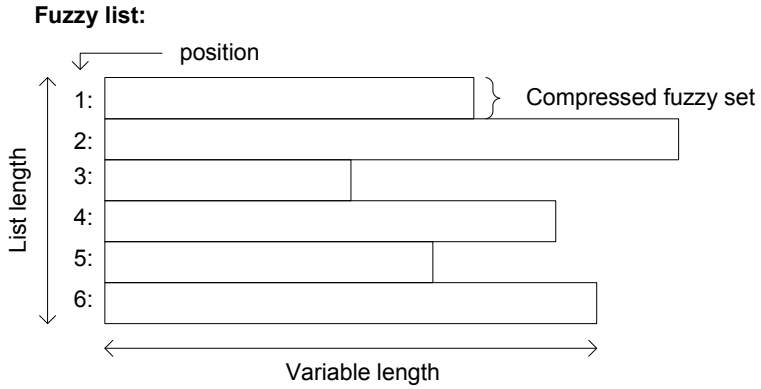


Figure 6.4: Fuzzy List Storage Organization

Previous work has already studied various options for effectively storing fuzzy sets. However, these storage options were studied in the context of fuzzy sets and their operators, e.g., intersection and union. We propose the Pattern Aware Word Aligned Run Length Encoding (PAWAH): a data compression scheme that allows arbitrary mathematical operations on the membership degrees. PAWAH is based on the PLWAH compression; simply put, PAWAH works on multiple bit patterns rather than on single bits.

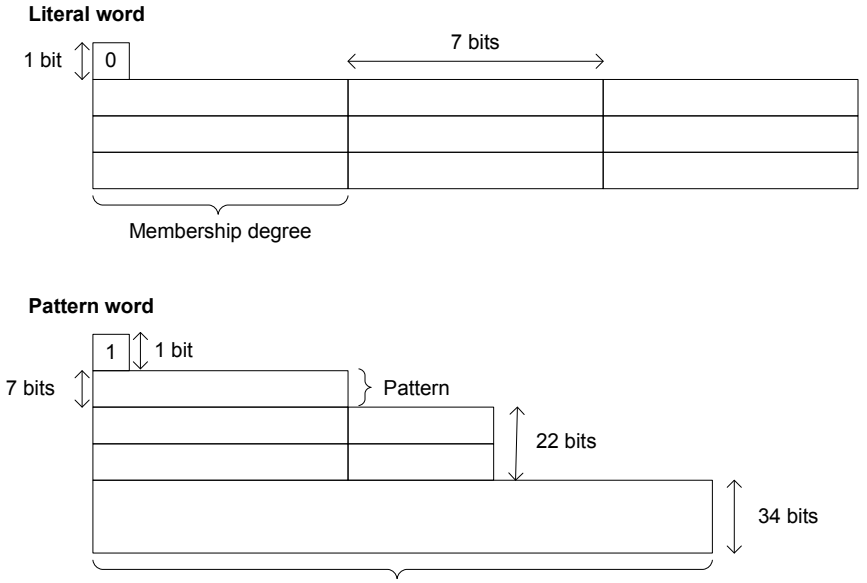


Figure 6.5: Literal and Pattern Words

6.B PAWAH Compression

We proceed by describing the PAWAH words and presenting the compression on an example.

PAWAH is composed of two types of words: fill words and literal words. As illustrated in Figure 6.5, literal and fill words are distinguished using their most significant bit (MSB). Literal words have their MSB unset. The remaining bits of a literal word contain values, each represented using multiple bits, e.g., if the values are ranging from 0 to 100, 7 bits are used to capture each value. The multibit values are appended to each other until no new value can fit in the current literal word. When the word length is not a multiple of the length of the value, the last bits are unused to respect the word alignment. A new literal word is used to store the remaining multibit values.

Fill words have their MSB set. Fill words are used to capture continuous sequences of words having an identical multibit values. The multibit values repeated among the words is represented using the bits following the MSB. These bits are referred to as the bit pattern of the fill word. The number of words that are represented by the bit patterns is captured by a counter located at the end of the word. The remaining bits located between the fill type and the counter are used to represent a literal word that directly follows the fill word and only differs from the fill word by

a few multibit values. Each multibit value is represented with its relative position within the literal word.

For example, let the following sequence of values ranging from 0 to 100 be compressed using PAWAH: $50 * 10, 5 * 100, 2 * 10$. Assuming a 64 bits word alignment, the compression is performed as follows. The compression steps are illustrated in Figure 6.6.

1. The uncompressed sequence of values is divided into groups of equal size, corresponding to the word length of the CPU architecture minus one. Thus, in the example, the groups have a size of 63 bits. Since each value can be represented with 7 bits, groups are composed of 9 values. The last group is padded with arbitrary values to be fully filled, zeros in the example.

2. Identical adjacent groups composed of 9 identical values are merged. In the example, the first group is a candidate for a merge, since it is exclusively composed of value 0001010 and followed by 4 identical groups, thus forming a total of 5 groups. The five groups are represented using a single group and a counter set to 5. The sixth and seventh groups are not composed of identical values and can therefore not be merged.

3. The groups are encoded in words. An additional bit is appended to the groups at the position of their Most Significant Bit (MSB). Merged groups are transformed into *fill words*. In the example, the last 34 Least Significant Bits (LSBs) are used to represent the number of merged groups each fill word contains. Thus, the first group becomes a fill word; its MSB is set and its counter is set to five, which corresponds to the number of merged groups and not the total number values. An extra unset bit is added as MSB to the groups that were not merged. Encoded words having their MSB unset are referred to as *literal words*. In our example, the second and third groups are transformed into literal words; each starts with an unset bit.

4. Literal words immediately following a fill word and having up to two different values from the fill word are identified. The relative positions of the differing values are calculated and are placed in the preceding fill word. The unused bits located between the fill word type bit and the counter bits are used for this purpose. In our example, 34 bits are used for the counter, and 9 bits are used for the representing the word type and the pattern. We thus have 22 bits remaining, namely the 8th to the 30th MSB. Seven bits are requires to represent the value. The relative position of the value of the word can be captured using $\lfloor \log_2 9 \rfloor = 4$ bits. Therefore, the value and its relative position take 11 bits, thus allowing 2 values and relative positions pairs to be stored using the 22 available bits remaining. In the example, the second word follows a fill word

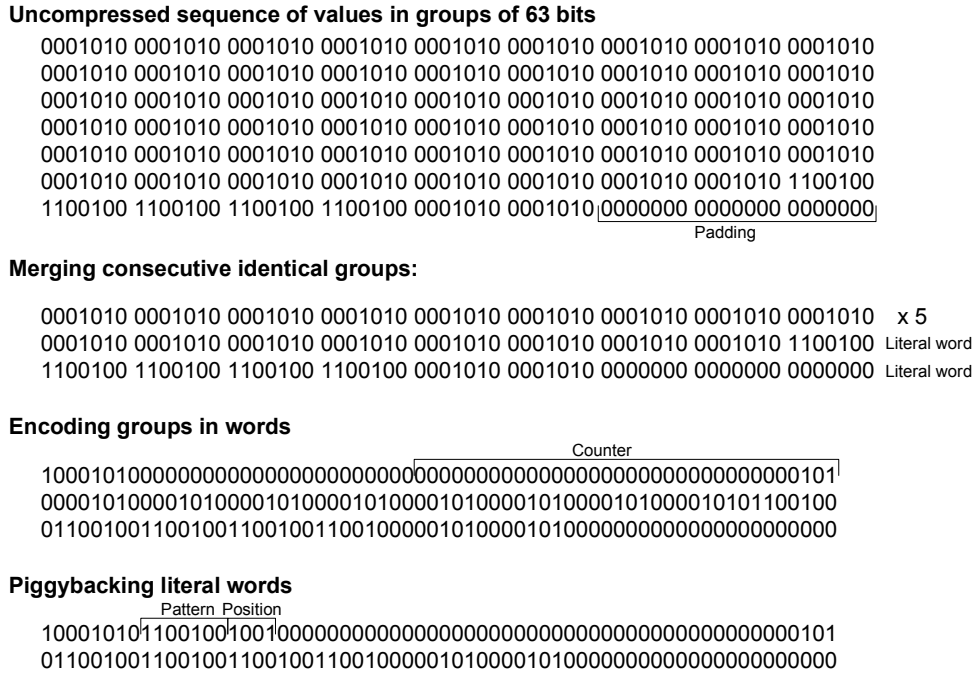


Figure 6.6: PAWAH Compression Example

and differ from it by only one value. The value and its position are placed into the fill word. In this way, the literal word is piggybacked by its preceding fill word. The last word cannot be piggybacked by its predecessor and is left as it is.

6.C PAWAH Compressed Fuzzy Set

A fuzzy set can be represented with a PAWAH compressed data: each element is identified by a given position in the sequence and each membership degree is captured with the value. The previous example would thus represent a fuzzy set where elements 1 to 50 have a membership degree of 10, elements 51 to 55 have a membership degree of 100, and elements 56 and 57 have a membership degree of 10.

The fuzzy sets can thus be fully represented with a single PAWAH data structure. This eases the implementation of mathematical operators on the membership degrees as a value is represented by only one word. For example, the sum operator can be implemented by a few simple bit manipulations as presented in Algorithm 6.1. The

extraction of a value from a word is performed by masking the adequate bits of the word.

Algorithm 6.1 PAWAH sum

word(word a**, word** b**)**
1: $c = v1(a) + v1(b)$
2: $c| = v2(a) + v2(b)$
3: $c| = v3(a) + v3(b)$
4: $c| = v4(a) + v4(b)$
5: $c| = v5(a) + v5(b)$
6: $c| = v6(a) + v6(b)$
7: $c| = v7(a) + v7(b)$
8: $c| = v8(a) + v8(b)$
9: $c| = v9(a) + v9(b)$
10: **return** c

Similarly, minimum and maximum functions can be performed word by word. The implementation of operators such as intersection and union between fuzzy sets only requires one reading of each input. Algorithm 6.2 presents the implementation of the computation of the minimum.

Algorithm 6.2 PAWAH minimum

word(word a**, word** b**)**
1: $c = min1(a, b)$
2: $c| = min2(a, b)$
3: $c| = min3(a, b)$
4: $c| = min4(a, b)$
5: $c| = min5(a, b)$
6: $c| = min6(a, b)$
7: $c| = min7(a, b)$
8: $c| = min8(a, b)$
9: $c| = min9(a, b)$
10: **return** c

6.D Operators

In this section, we propose the implementation of three operators defined for fuzzy lists: unary aggregation, binary aggregation, and projection. Most operators can be defined by a combination of these three basic operators.

6.D.1 Mapping and Aggregation Structures

The mapping is defined by a list of destination positions. As illustrated in Figure 6.7, for each destination position, the mapping structure points to the fuzzy sets defined at

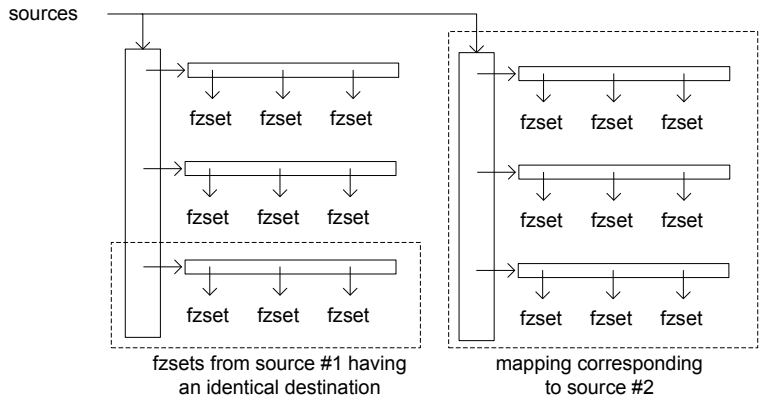


Figure 6.7: Mapping Data Structure

the position of origin. The aggregation is performed between all the elements that are moved to an identical position as defined by the mapping. To manipulate groups of elements having an identical position, we rely on previously developed data structures for manipulating fuzzy sets. Implementations details of the mapping and aggregation structures are presented in Algorithms 6.3 and 6.4.

Algorithm 6.3 Mapping structure

```
struct mapping_struct {
    int * nr_source;
    source * sources;
} typedef * mapping;

struct source_struct {
    int source_length;
    fzset_array * fzsets;
} typedef * source;

struct fzset_array_struct {
    int nr_fzset;
    fuzzyset * fzset;
} typedef * fzset_array;
```

6.D.2 Unary Aggregation

A unary aggregation requires a mapping definition and an aggregation function.

Algorithm 6.4 Aggregation structure

```
struct aggregation_struct {
    agg_state * state;
    void *(*init)(aggregation a);
    void *(* step)(fuzzyset *, agg_state *);
    void *(* finalize)(void *);
} typedef * aggregation;
```

The implementation is done in two steps. First, the elements and their corresponding membership degree are moved. Second, the elements located at identical positions are aggregated. The implementation of a moving average aggregation is presented below. The unary aggregation is implemented as follows:

- A internal state captures all temporary information that is needed between each iteration. For a moving average, we are storing the list of all the elements and their corresponding membership values. This list is represented using a fuzzy set. Additionally, since we are performing an average, the internal state has to keep a counter of the number of aggregated positions.
- An initialization function defines the initial state. In this case, we define a empty fuzzy set with a counter set to zero.
- A step function performs one iteration of the aggregation. The aggregation is done between the elements located at the same position. In our example, we calculate the new mean and increment the counter of the number of position merged.
- A finalization function transforms the last internal state into the final fuzzy set for a given position. In our example, we remove the counter of iterations from the state.

Algorithm 6.5 presents the pseudo C code of the implementation of the generic unary aggregation. The aggregation structure is a generic container for all aggregation functions. In order to perform the average presented in the example, $step$ has to point to the average function. The $+$ and $/$ operators on fuzzy sets are assumed to be defined to respectively sum and divide the membership degree of each element of the fuzzy set.

6.D.3 Projection

The projection operator is similar to the unary aggregation but is operating on the elements rather than on the position. The projection operator is defined with a projection mapping that transforms new elements from a set of old elements. The membership

Algorithm 6.5 Unary aggregation

```
    // x: input fuzzy list
    // z: output fuzzy list
    void unary(mapping m, aggregation a, fuzzylist x, fuzzylist z)
    // i: destination of the position mapping
    // m[i]: set of fuzzy sets moved to position i
 1: for (int i = 0; i < size(m); i++) do
 2:     a -> state = (a -> init)(a); // mem alloc
 3:     for all fset in m[i] do
 4:         a -> state = (a -> step)(fset, a -> state);
 5:     z[i] = (a -> finalize)(a -> state);


    void average(fuzzyset fzset, agg_state *a)
 1: a -> counter++;
 2: a -> fzset = a -> fzset + (fzset - a -> fzset)/a -> counter;
```

degrees of the old elements are aggregated to generate the new element membership degree. The new membership degree can be generated by multiple membership degrees corresponding to the input elements.

Algorithm 6.6 Projection

```
    void project(projection p, aggregation a, fuzzylist x, fuzzylist z)
    // for each fuzzy set of the fuzzy list
 1: for all pos in x do
 2:     fzset = getfzset(x, pos);
 3:     for all el in proj do
 4:         tmp: new temporary fuzzy set
 5:         for all oldel of proj[el] do
 6:             oldmu = getmu(fzset, oldel)
 7:             newmu = (p-> step)(oldmu);
 8:         addfzelem(tmp, el, newmu)
 9:     z[pos] = tmp;
```

6.D.4 Binary Aggregation

Binary aggregation deals with repositioning fuzzy sets. The mapping is defined for each destination as a list of source position and source fuzzy list. The fuzzy sets are aggregated with respect to their position.

Algorithm 6.7 Binary aggregation

```
    struct aggregation_struct {
        void * state;
        void *(*init)(aggregation a);
        void *(* step)(fuzzyset *, agg *);
        void *(* finalize)(void *);
    } typedef aggregation;

    // x: input fuzzy list
    // y: input fuzzy list
    // z: output fuzzy list
    void binary(mapping m, aggregation a, fuzzylist x, fuzzylist y, fuzzylist z)
    // i: destination of the position mapping
    // m[i]: set of fuzzy sets from x and y moved to position i
1:  for (int i = 0; i < size(m); i++) do
2:      a → state = (a → init)(a); // mem alloc
3:      for all fset_x and fset_y in m[i] do
4:          a → state = (a → step)(fset_x, a → state);
5:          a → state = (a → step)(fset_y, a → state);
6:      z[i] = (a → finalize)(a → state);

    void average(fuzzyset fzset, agg_state *a)
1:  a → counter++;
2:  a → fzset = a → fzset + (fzset − a → fzset)/a → counter;
```

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The growth of digital music collections and the tremendous gain in popularity of music applications have issued new challenges to music data management systems. This thesis reports on the design of a Music Warehouse (MW), a dedicated data warehouse optimized for the management of music content. It presents concepts to facilitate the representation and manipulation of music information. Additionally, it provides effective music data management tools designed to support modern music applications, such as online music recommendation systems and music information retrieval systems.

The thesis follows the elaboration of a Music Warehouse. It comprises four major interdependent layers. First, an exploratory phase defines a Music Warehouse, identifies challenges, and proposes an approach for addressing some of them. Second, a data collection phase presents methods for gathering music features and playlists. Third, the data organization and representation are studied for each type of the collected music information. Fourth, dedicated data manipulation tools are developed. The thesis structure is sketched in Figure 7.1. The four layers are covered by five chapters, they are summarized below.

The thesis begins by laying the necessary foundations for the development of MWs. Inspired by the previous successes of data warehouses in business integration issues and on-the-fly analytical demands, MWs are described as centralized data stores, based on the data warehousing approach, that are optimized to answer the fast-search needs of large music information retrieval systems. Chapter 2 proposes to organize music metadata into four categories and presents a case study of musical database management. The usage scenario, the system architecture, and the preliminary research ideas described in Chapter 2 are pursued and constitute the topics of Chapters 4 and 6. Chapter 2 is thus an essential chapter that brings to light the con-

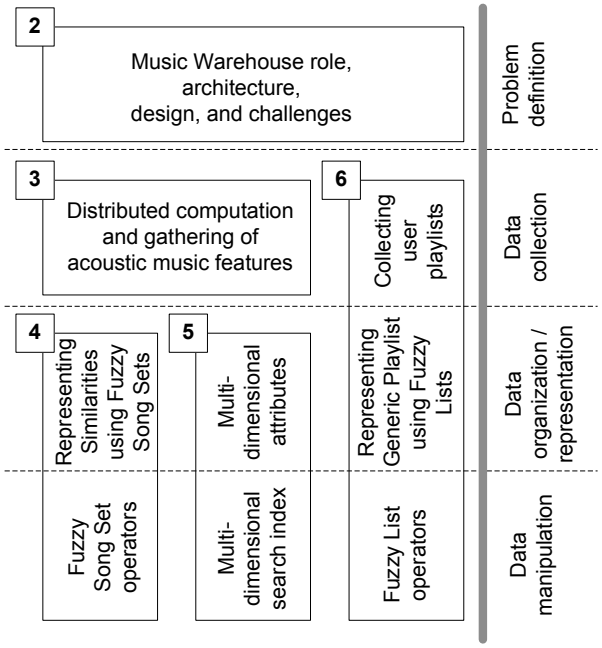


Figure 7.1: Thesis Structure

necting thread between MW and new concepts, such as Fuzzy Song Sets and Fuzzy Lists. Additionally, Chapter 2 proposes ten promising research challenges for MW. While some of these challenges remain unaddressed, the thesis identifies them in the hope that future research in these directions will be conducted. Most of the proposed research topics would benefit a wide range of data warehousing applications.

Next, driven by a voracious appetite for music data, the thesis addresses issues bound to the gathering of large amounts of music features. The automatic extraction of high-level features over a large music collection suffers from a major scalability issues: feature extraction is a computationally expensive process. Attempts to spread the feature extraction among a large number of publicly available computers have failed due to the copyright restrictions, which limit the exchange of the audio content. Chapter 3 introduces an efficient framework for extracting high-level audio features. The distributed extraction is performed in a two-step process in order to respect copyright. The proposed framework has successfully demonstrated its ability to efficiently extract high-level features on a large music collection. Furthermore, the framework proves to be efficient and flexible for performing similarity searches using the extracted features. This is done by allowing users to constrain the search within a range and specify a weighted combination of high-level features. To optimize the

search, three different approaches are compared in terms of query time and storage. The challenge lies in the ability to accommodate efficient search for both absolute and relative types of similarity measures. Tools for manipulating relative similarity measures and absolute similarity measures are presented in Chapters 4 and 5 respectively.

Fuzzy sets prove to be well suited for addressing various problematic scenarios commonly encountered in recommendation systems. In Chapter 4, after defining the concept of Fuzzy Song Sets and presenting an algebra to manipulate them, we demonstrate the usefulness of Fuzzy Song Sets and their operators to handle various information management scenarios in the context of a music warehouse. For this purpose, we create two multidimensional cubes: the Song Similarity Cube and the User Feedback Cube. Three data storage options, arrays, tables and WAH compressed bitmaps, are envisioned for representing Fuzzy Song Sets. The impact of these data structures on the storage space and operator performance is then discussed. With respect to storage, while arrays first show to be a very good choice from a theoretical point of view, they suffer from a significant overhead. Estimates taking into account DBMS overheads show that the differences between WAH bitmaps and arrays vanish as the number of elements grows. The different data organizations in WAH bitmaps and in arrays cause operators to behave very differently depending on the number of elements. Arrays are very efficient when the number of elements remains limited. However, arrays behave poorly for larger sets. Requiring a more complex management, bitmaps suffer from a higher starting overhead, which is mostly visible when the number of elements is low. As the number of elements grows, operations on bitmap are faster than on arrays. In our experiment with the largest number of elements, the Union operator performs 5 times faster on WAH bitmaps than on arrays; the speedup factor is 7 for the Top operator and 85 for the Reduce operator.

Absolute similarity measures are characterized by high dimensionality. Chapter 5 improves the performance of high-dimensionality range queries. We present Position List Word Aligned Hybrid (PLWAH), a bitmap compression scheme that compresses sparse bitmaps better and answers long range queries faster than WAH, which was so far recognized as the most efficient bitmap compression scheme for high cardinality attributes. The storage gain essentially varies with the following parameters: the size of the data set, the attribute distribution, the attribute cardinality, and the word length. For uniformly distributed high cardinality attributes, we both theoretically prove and experimentally observe that the compression ratio is twice as good as for WAH. In terms of performance, PLWAH and WAH are comparable for short range queries. However, for long range queries, PLWAH is faster than WAH.

Generic playlists offer a pragmatic answer to the need for playlist management tools that has arisen in online music communities. Generic playlists respect both the consensual and the subjective nature of playlists by defining for each position in the

sequence a likelihood degree of the presence of a given song. As illustrated by a concrete scenario, generic playlists are a flexible and concrete solution for constructing, sharing and personalizing playlists. Chapter 6 provides a solid foundation for the development of generic playlists by formally defining the concept of Fuzzy Lists and their algebra. Examples of generic playlists motivate the use of the presented Fuzzy Lists operators. Three basic operators are proposed. Their similarity with relational algebra operators facilitates their implementation in database systems. Additionally, a prototypical implementation of the Fuzzy Lists is presented. This implementation captures Fuzzy Lists as lists of fuzzy sets. Each fuzzy set is represented using a new compression technique facilitating arithmetic on the membership degrees of the elements. The pseudo code of some of the fundamental Fuzzy List operators is provided.

The main contributions presented in the thesis are enumerated below.

1. We propose a scalable framework to extract music features. The framework distributes the extraction of the music features to client nodes in order to compute the similarity measures without violating copyright.

2. We propose the concept of Fuzzy Song Sets for capturing music information and an algebra for facilitating their manipulation. Three usage scenarios and the corresponding multidimensional cubes extended with Fuzzy Song Sets are described to illustrate the usage of Fuzzy Song Sets in MW.

3. We implement common fuzzy set operators using WAH and array compression of Fuzzy Song Sets and prove that fuzzy sets represented with WAH compressed bitmaps are the most efficient to manipulate large fuzzy sets.

4. We propose PLWAH, a new bitmap compression scheme that reduces the size of bitmaps and improves the computation speed of bitwise operators. PLWAH performs searches on highly multidimensional data very efficiently. PLWAH is useful to handle the multidimensional range queries commonly used for dealing with music features.

5. We define the concept of Fuzzy Lists and propose an algebra to manipulate them. Fuzzy Lists are used to represent, manipulate, and share playlists.

6. We propose a storage representation and the corresponding implementation of the operators of Fuzzy Lists. Fuzzy Lists are stored in a compressed format. The new compression, referred to as PAWAH, allows performing arithmetic operations on the membership degree directly on the compressed representation.

The thesis presents innovative concepts and techniques to deal with large amounts of music features. For example, new concepts, such as Fuzzy Song Sets and Fuzzy Lists, allow organizing and manipulating music similarities and users musical preferences. New techniques, such as the PLWAH and PAWAH compressions schemes, improve the multidimensional range queries commonly used to search among music features. They also improve Fuzzy Song Sets and Fuzzy Lists storage and the performance of their operators. The concepts and techniques were theoretically and experimentally studied using a large collection of music features computed thanks to the scalable music similarity computation framework described in Chapter 3.

While the presented research was motivated by the development of an MW, some of the contributions, e.g., PLWAH, PAWAH and Fuzzy Lists, are easily transposable to other application domains.

7.2 Future Work

As illustrated in Figure 7.2, future work can be pursued in three directions: first, towards bringing further the technical aspects of the presented concepts and methods, second, towards facilitating the interactions between the developed tools and other existing systems, and third, towards widening the scope of applications by generalizing the contributions to other domains.

The presented contributions would benefit from the following studies and technical improvements. The distributed framework for the extraction of music features would gain from an automatic client update system. The system would be able to notify the clients when new or updated extraction methods are available. The clients would then download and use the most recent extraction algorithms. Fuzzy Song Sets should benefit from the proposed PLWAH and PAWAH compression schemes. Empirical studies and comparisons between PLWAH and PAWAH compressed Fuzzy Song Sets have to be conducted. Future work also encompasses studying the performance impact of PLWAH on complementary bitmap indexing strategies. A promising research direction consists in the development of primary index that builds aggregated bitmaps based on query patterns, frequencies, and data distribution. The primary index would be able to select the most efficient aggregation path, the required bitwise operations, and the type of bitwise algorithm to use. Finally, analytical and empirical studies of the storage requirements and operator performances of PAWAH compressed Fuzzy Lists are needed. Further studies would facilitate the identification of possible weaknesses of the PAWAH compression scheme and provide insights to effective optimizations.

A second future work direction is to facilitate the interactions with other music information retrieval platforms and to solve portability and integration issues linked to different data representation and system architectures. The presented distributed mu-

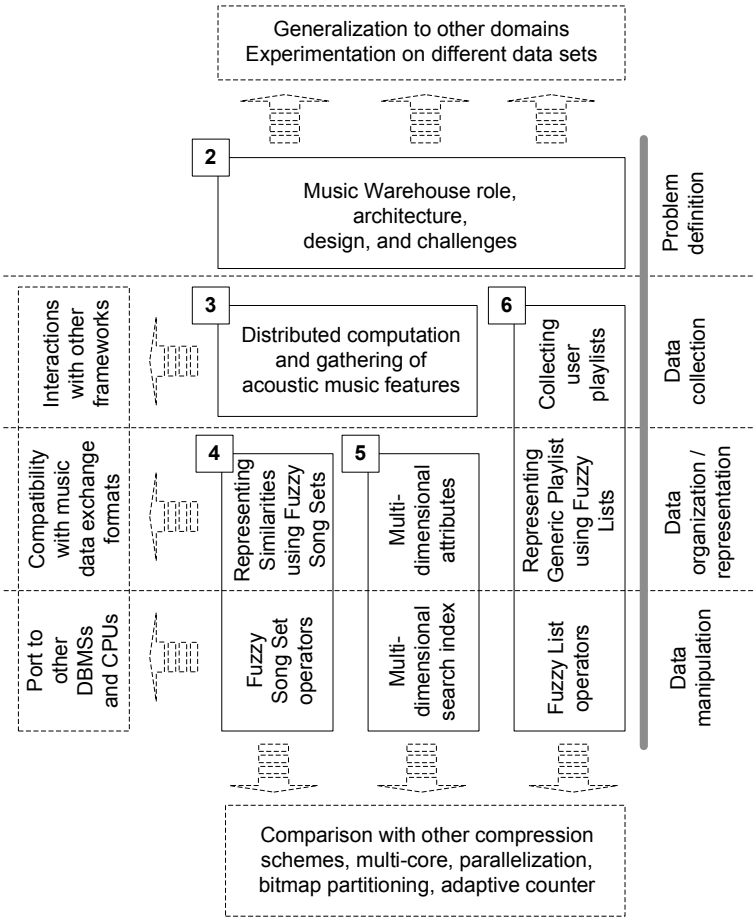


Figure 7.2: Future Work

sic feature extraction framework should interact with the Networked Environment for Music Analysis (NEMA) [34]. In this perspective, work on compatibility with new data formats, such as ACE/XML [61], should be pursued. Fuzzy sets storage representations and operators should be used in music frameworks, in particular those relying on fuzzy sets. Porting the PLWAH and PAWAH bitmap compression schemes to other DBMSs would widen the audience and facilitate the integration to other systems. Similarly, there is the need to adapt the compression scheme to different CPU architectures. For example, PLWAH words have a short counter on 32 bit long CPU architecture. This could become problematic on extremely sparse bitmaps where runs become very long. The adaptive counter is a promising approach that requires an experimental evaluation. Furthermore, the PLWAH compression scheme could take advantage of multi-core CPUs. Various techniques of parallelization should be

studied. In particular, the partitioning of the data index is an interesting direction that eliminates the need for critical sections and limits the maximum run-length. It thus constitutes a potential alternative to the issue of too short counters in the case of very long runs.

The contributions could be generalized in order to bring benefits to other domains. Such generalization includes comparing the extraction framework of music features with other generic frameworks such as BOINC [5]. A conceptual comparison with the Map-Reduce programming model [28] would also be beneficial, in particular on the issue of assigning priorities to certain tasks of the distributed computation. Fuzzy Sets and Fuzzy Lists representations can be used in other domains; studies of the impact of their respective data representations should be conducted in other domains than multimedia. The concept of aggregation between Fuzzy Lists needs to be further developed in order to build fuzzy lists hierarchies. PLWAH can be applied directly as a replacement to existing bitmap compression techniques. However, further analytical studies of the PLWAH compression scheme on different data distribution need to be conducted in order to demonstrate the effectiveness of PLWAH in different domains. Similarly, PLWAH would benefit from additional comparisons with other bitmap compression algorithms.

In summary, the contributions presented in this thesis constitute a solid foundation for developing an MW. The concepts are flexible enough to support a wide spectrum of music applications. The techniques are efficient enough to address scalability issues related to large music collections and high numbers of requests. While the contributions are significant and the MW has already demonstrated its robustness, many future improvements remain desirable. Three main future work directions are identified: improvement of the presented concepts and techniques; compatibility with other MIR tools and systems; and generalization and application of the contributions to new domains.

Bibliography

[1] Meandre documentation. http://seasr.org/meandre/documentation/.

[2] Advanced Micro Devices. *Software Optimization Guide for AMD Family 10h Processors*, November 2008.

[3] M. Alghoniemy and A. Tewfik. A network flow model for playlist generation. In *Proceedings of the IEEE International Conference on Multimedia and Expo, ICME*, pages 445–448, 2001.

[4] D. P. Anderson. The Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu/.

[5] D. P. Anderson. BOINC: a system for public-resource computing and storage. In *Proceedings of the International Workshop on Grid Computing*, pages 4–10, 2004.

[6] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[7] G. Antoshenkov. US Patent #5363098: Byte aligned data compression. 1994.

[8] G. Antoshenkov and M. Ziauddin. Query processing and optimization in ORACLE RDB. *The International Journal on Very Large Data Bases, VLDB*, 5(4):229–237, 1996.

[9] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate Encoding for direct access and query processing over compressed bitmaps. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 846–857, 2006.

[10] J. Aucouturier and F. Pachet. Music similarity measures: What's the use? In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 157–163, 2002.

[11] J. Aucouturier and F. Pachet. Representing musical genre: A state of the art. *Journal of New Music Research*, 32(1):83–93, 2003.

[12] J.-J. Aucouturier and F. Pachet. Scaling up music playlist generation. In *Proceedings of the IEEE International Conference on Multimedia and Expo, ICME*, pages 105–108, 2002.

[13] J.-J. Aucouturier, F. Pachet, and M. Sandler. The way it sounds : Timbre models for analysis and retrieval of polyphonic music signals. *IEEE Transactions of Multimedia*, 7(6):1028–1035, 2005.

[14] T. L. Blum, D. F. Keislar, J. A. Wheaton, and E. H. Wold. Method and article of manufacture for content-based analysis, storage, retrieval, and segmentation of audio information. 1999.

[15] G. Bordogna, D. Lucarella, and G. Pasi. A fuzzy object oriented data model. In *Proceedings of the IEEE Conference on Fuzzy Systems*, pages 313–318, 1984.

[16] K. Bosteels and E. E. Kerre. Evaluating and analysing dynamic playlist generation heuristics using radio logs and fuzzy set theory. Submitted for publication.

[17] K. Bosteels and E. E. Kerre. A fuzzy framework for defining dynamic playlist generation heuristics. Submitted for publication.

[18] K. Bosteels and E. E. Kerre. Fuzzy audio similarity measures based on spectrum histograms and fluctuation patterns. In *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering*, pages 361–365, 2007.

[19] D. Brackett. *Interpreting popular music*. Cambridge University Press, 1995.

[20] S. Bray and G. Tzanetakis. Distributed audio feature extraction for music. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 434–437, 2005.

[21] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Record*, 27(2):355–366, 1998.

[22] B. Y. Chua. *Automatic Extraction of Perceptual Features and Categorization of Music Emotional Expression from Polyphonic Music Audio Signals*. PhD thesis, Monash University, Victoria, Australia, 2007.

[23] B. Y. Chua and G. Lu. Improved perceptual tempo detection of music. In *Proceedings of the International Multimedia Modeling Conference, MMM*, pages 316–321, 2005.

[24] B. Y. Chua and G. Lu. Perceptual rhythm determination of music signal for emotion-based classification. In *Proceedings of the International Multimedia Modeling Conference, MMM*, pages 4–11, 2006.

[25] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 426–435, 1997.

[26] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[27] W. W. Cohen and W. Fan. Web-collaborative filtering: recommending music by crawling the Web. *Computer Networks*, 33(1–6):685–698, 2000.

[28] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementationthe, OSDI*, pages 137–150, 2004.

[29] F. Deliège and T. B. Pedersen. Music warehouses: Challenges for the next generation of music search engines. In *Proceedings of the 1st Workshop on Learning the Semantics of Audio Signals, LSAS*, pages 95–105, 2006.

[30] F. Deliège and T. B. Pedersen. Using fuzzy song sets in music warehouses. In *Proceedings of the 8th International Conference on Music Information Retrieval, ISMIR*, pages 21–26, 2007.

[31] B. A. Devlin and P. T. Murphy. An architecture for a business and information system. *IBM Systems Journal*, 27(1):60–80, 1988.

[32] C. Digout and M. A. Nascimento. High-dimensional similarity searches using a metric pseudo-grid. In *Proceedings of the International Conference on Data Engineering Workshop, ICDE Workshop*, pages 1174–1184, 2005.

[33] J. S. Downie. The music information retrieval evaluation exchange (MIREX). *D-Lib Magazine*, 12(12):795–825, December 2006.

[34] J. S. Downie and I. Fujinaga. NEMA: The networked environment for music analysis. http://nema.lis.uiuc.edu/.

[35] J. S. Downie, J. Futrelle, and D. K. Tcheng. The international music information retrieval systems evaluation laboratory: Governance, access and security. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 9–15, 2004.

[36] J. S. Downie and M. Nelson. Evaluation of a simple and effective music information retrieval method. In *Proceedings of the ACM International Conference on Research and Development in Information Retrieval, SIGIR*, pages 73–80, 2000.

[37] A. F. Ehmann, J. S. Downie, and M. C. Jones. The music information retrieval evaluation exchange "Do-It-Yourself" Web Service. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 323–324, 2007.

[38] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 4th Edition*. Addison-Wesley Longman Publishing Co., 2003.

[39] J. Evans. *A Scalable Concurrent malloc Implementation for FreeBSD*. FreeBSD, April 2006.

[40] F. Fabbri. Browsing music spaces: Categories and the musical mind. http://www.mediamusicstudies.net/tagg/xpdfs/ffabbri9907us.pdf, 1999.

[41] J. Galindo, M. Piattini, and A. Urrutia. *Fuzzy Databases: Modeling, Design and Implementation*. Idea Group Pub, 2005.

[42] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith. Query by humming: Musical information retrieval in an audio database. In *Proceedings of the ACM Multimedia*, pages 231–236, 1995.

[43] K.-S. Goh, B. Li, and E. Chang. DynDex: a dynamic and non-metric space indexer. In *Proceedings of the ACM Multimedia*, pages 466–475, 2002.

[44] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[45] W. T. Hicken. US Patent #20060265349A1: Sharing music essence in a recommendation system. 2006.

[46] W. H. Inmon. *Building the Data Warehouse*. Wiley Computer Publishing, 2 edition, 1996.

[47] Intel. *Intel SSE4 Programming Reference*, July 2007.

[48] International Organisation for Standardisation: ISO/IEC JTC1/SC29/WG11N6828. Mpeg-7 overview (version 10), 2004.

[49] C. A. Jensen, E. M. Mungure, T. B. Pedersen, and K. Sørensen. A data and query model for dynamic playlist generation. In *Proceedings of the IEEE*

International Workshop on Multimedia Databases and Data Management, IEEE-MDDM, pages 65–74, 2007.

[50] H. Jensen, D. P. W. Ellis, M. G. Christensen, and S. H. Jensen. Evaluation of distance measures between Gaussian mixture models of MFCCs. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 107–108, 2007.

[51] I. Karydis, A. Nanopoulos, A. Papadopoulos, D. Katsaros, and Y. Manolopoulos. Content-based music information retrieval in wireless ad-hoc networks. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 137–144, 2005.

[52] I. Karydis, A. Nanopoulos, A. Papadopoulos, and Y. Manolopoulos. Audio indexing for efficient music information retrieval. In *Proceedings of the International Multimedia Modeling Conference, MMM*, pages 22–29, 2005.

[53] R. Kimball, L. Reeves, M. Ross, and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*. Wiley, 1996.

[54] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 215–226, 1996.

[55] N. Koudas. Space efficient bitmap indexing. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, 2000.

[56] T. Lehn-Schiler, J. Arenas-Garca, K. B. Petersen, and L. K. Hansen. A genre classification plug-in for data collection. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 320–321, 2006.

[57] C. E. Leiserson, H. Prokop, and K. H. Randall. Using de Bruijn sequences to index a 1 in a computer word, 1998. Available at http://supertech.csail.mit.edu/papers.html.

[58] B. Logan and A. Salomon. A music similarity function based on signal analysis. In *Proceedings of the International Conference on Multimedia and Expo, ICME*, pages 745–748, 2001.

[59] D. Lübbers. SoniXplorer: Combining visualization and auralization for content-based exploration of music collections. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 590–593, 2005.

[60] M. Mandel and D. Ellis. Song-level features and support vector machines for music classification. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 594–599, 2005.

[61] C. McKay and I. Fujinaga. Combining features extracted from audio, symbolic and cultural sources. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 597–602, 2008.

[62] C. McKay, D. McEnnis, and I. Fujinaga. Overview of OMEN. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 7–11, 2006.

[63] R. Neumayer, M. Dittenbach, and A. Rauber. PlaySOM and PocketSOMPlayer, alternative interfaces to large music collections. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 618–623, 2005.

[64] P. O'Neil. Model 204 architecture and performance. In *Proceedings of the Workshop in High Performance Transaction Systems*, pages 40–59, 1987.

[65] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 38–49, May 1997.

[66] F. Pachet. Musical metadata and knowledge management. In *Encyclopedia of Knowledge Management*, pages 672–677. Idea Group, 2005.

[67] F. Pachet and D. Cazaly. A taxonomy of musical genre. In *Proceedings of the Content-Based Multimedia Information Access Conference*, volume 2, pages 1238–1246, 2000.

[68] E. Pampalk. Speeding up music similarity. In *Proceedings of the Music Information Retrieval Evaluation eXchange, MIREX*, 2005.

[69] E. Pampalk, T. Pohle, and G. Widmer. Dynamic playlist generation based on skipping behavior. In *Proceedings of the 4th International Conference on Music Information Retrieval, ISMIR*, pages 634–637, 2005.

[70] S. Pauws and B. Eggen. PATS: Realization and user evaluation of an automatic playlist generator. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 179–192, 2002.

[71] T. B. Pedersen and C. S. Jensen. Multidimensional data modeling for complex data. In *Proceedings of the 15th International Conference on Data Engineering, ICDE*, page 336, Washington, DC, USA, 1999. IEEE Computer Society.

[72] T. B. Pedersen and C. S. Jensen. Multidimensional database technology. *Computer*, 34(12):40–46, 2001.

[73] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. A foundation for capturing and querying complex multidimensional data. *Information Systems*, 26(5):383–423, 2001.

[74] T. Pohle, E. Pampalk, and G. Widmer. Generating similarity-based playlists using traveling salesman algorithms. In *Proceedings of the 8th International Conference on Digital Audio Effects, DAFx*, pages 220–225, 2005.

[75] H. Prade and C. Testemale. Generalizing database relational algebra for the treatment of incomplete or uncertain information and vague queries. *Information Sciences*, 34:115–143, 1984.

[76] M. Rafanelli. *Multidimensional Databases: Problems and Solutions*. Idea Group, 2003.

[77] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. *SIGMOD Record*, 30(2):47–57, 2001.

[78] D. Rotem, K. Stockinger, and K. Wu. Optimizing candidate check costs for bitmap indices. In *Proceedings of the ACM Conference on Information and Knowledge Management, CIKM*, pages 648–655, 2005.

[79] W. B. Rubenstein. A database design for musical information. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 479–490, 1987.

[80] M. M. Ruxanda and C. S. Jensen. Efficient similarity retrieval in music databases. In *Proceedings of the International Conference on Management of Data, COMAD*, pages 56–67, 2006.

[81] T. Seidl and H. Kriegel. Optimal multi-step K-nearest neighbor search. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 154–165, 1998.

[82] S. Sheu and J.-R. Wu. GC*-tree: a generic index for perceptual similarity search. In *Proceedings of the 3rd International Conference on Information Technology: Research and Education, ITRE*, pages 167–171, 2005.

[83] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Bringing order to query optimization. *SIGMOD Record*, 31(2):5–14, 2002.

[84] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on Run-Length and Huffman encoding. *To appear in Information Systems*, 2009.

[85] K. Stockinger and K. Wu. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, chapter Bitmap Indices for Data Warehouses, pages 157–178. 2007.

[86] The PostgreSQL Global Development Group. PostgreSQL 8.2.0 documentation. In *http://www.postgresql.org/docs/manuals/*, 2006.

[87] E. Thomsen. *OLAP Solution: Building Multidimensional Information Systems*. Wiley, 1997.

[88] B. Tripathy and G. Pattanaik. On some properties of lists and fuzzy lists. *Information Sciences*, 168(1–4):9–23, 2004.

[89] G. Tzanetakis, G. Essl, and P. R. Cook. Automatic musical genre classification of audio signals. In *Proceedings of the International Conference on Music Information Retrieval, ISMIR*, pages 293–302, 2001.

[90] C. Wang, J. Li, and S. Shi. A music data model and its application. In *Proceedings of the 10th International Multimedia Modeling Conference, MMM*, pages 79–85, 2004.

[91] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensinal spaces. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 194–205, 1998.

[92] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions Database Syst.*, 31(1):1–38, 2006.

[93] K. Wu and P. Yu. Range-based bitmap indexing for high cardinality attributes with skew. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, pages 61–67, 1998.

[94] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for Data Warehouses. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 220–230, 1998.

[95] M. J. Wynblatt and G. A. Schloss. Control layer primitives for the layered multimedia data model. In *Proceedings of the ACM International Conference on Multimedia*, pages 167–177, 1995.

[96] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 201–208, 1998.

[97] C. Yu, B. Ooi, K. Tan, and H. Jagadish. Indexing the distance: An efficient method to KNN processing. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 421–430, 2001.

[98] L. A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Information and Control*, 8:338–353, 1965.

[99] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1997.

[100] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 59–71, 2006.

Summary in Danish

En ualmindelig vækst i mængden af digitalt musik kræver nye systemer og algoritmer til at organisere og søge i musik databaser. En lang række nye forskningsområder inden for sådanne musik databaser har gennem de seneste år set dagens lys. Denne afhandling fokuserer på de datatekniske problemstillinger, der vedrrer systemer til automatisk musik anbefalinger. Afhandling foreslår værktjer til udregning, repræsentation, manipulation, og søgning i musik lighedsmålinger.

Afhandlingen starter med at definere et brugsscenario, et Musik Warehouse koncept, en systemarkitektur, og et overblik over forskningsmæssige problemstillinger.

Afhandlingen fortsætter med at præsentere et framework som gør det muligt at udtrække musikegenskaber uden at krænke ophavsrettighederne for de involverede kunstnere. Frameworket bliver brugt til at skabe et datasæt af musik lighedsmåleinger. En række metoder bliver præsenteret, som forbedre lighedssøgningerne ved brug af vægtede-kombinationer over det indsamlede data.

For ligheden mellem par af musikstykker defineres Fuzzy Song Sets. Lagring og implementering af fundamentale operatorer studeres for forskellige interne repræsentationer. Studierne viser, at komprimerede bitmaps med en udvidet version af Word Aligned Hybryd algorithmen som intern repræsentation giver de bedste resultater.

Efterfølgende præsenteres de udfordringer, som findes i forhold til søgninger i et metrisk musik lighedsrum. Komprimerede bitmaps foreslås til multidimensionelle interval forespørgsler, og en ny bitmap komprimeringsalgoritme, Position List Word Aligned Hybrid (PLWAH), præsenteres. PLWAH giver forbedrede resultater i forhold til eksisterende komprimeringsalgoritmer, både i forhold til komprimeringsraten og i forhold til effektiviteten af operatorer.

Til sidst defineres Fuzzy Lists, som er matematiske objekter, der gør det muligt at repræsentere og manipulere afspilningslister. For at holde Fuzzy Lists konkrete, bliver operatorer defineret og illustreret igennem eksempler i konteksten af en afspilningsliste applikation. Der gives desuden en implementeringmentering af de fundamentale operatorer.

Som omsummering, ophandler denne afhandling altså de udfordringer, der findes i forhold til lagring og behandling af musik lighed. For hver udfordring gives konkrete

løsninger, der giver forbedringer over de eksisterende tilgængelige løsninger i både den akademiske og industrielle verden. Selvom fokus i denne afhandling er på det musikalske domæne, er de præsenterede teknikker generelle og kan i de fleste tilfælde finde brugbarhed inden for andre domæner, hvorved brugbarheden af de rapporterede resultater øges.